HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

# MASTERARBEIT

B.Sc.
**Christian Mahner**

## Entwicklung eines mikrocontrollerbasierten Synchronisationsverfahrens für niederfrequente, schmalbandige, analoge Kommunikationskanäle

2017

Fakultät **Ingenieurwissenschaften**

# MASTERARBEIT

# Entwicklung eines mikrocontrollerbasierten Synchronisationsverfahrens für niederfrequente, schmalbandige, analoge Kommunikationskanäle


Autor:
**Christian Mahner**

Studiengang:
Master Elektrotechnik

Seminargruppe:
ET15s1-M

Erstprüfer:
Prof. Dr. Dr.-Ing. Hartmut Luge

Zweitprüfer:
Prof. Dr.-Ing. Christian Schulz


Mittweida, 2017

**Bibliografische Angaben**

Mahner, Christian: Entwicklung eines mikrocontrollerbasierten Synchronisationsverfahrens für niederfrequente, schmalbandige, analoge Kommunikationskanäle, 119 Seiten, 45 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Ingenieurwissenschaften

Masterarbeit, 2017

**Referat**

Diese Masterarbeit beschäftigt sich mit der Entwicklung eines mikrocontrollergestützten Verfahrens zur Synchronisation von Teilnehmern, welche über niederfrequente, schmalbandige Analogkanäle kommunizieren. Es werden eine mögliche Anwendung, sowie verschiedene Lösungsansätze aufgezeigt. Weiterhin wird anhand eines entwickelten Laborversuches dargelegt, wie eines der vorgestellten Verfahren implementiert werden kann und welche Ergebnisse damit erzielt wurden.

# I. Inhaltsverzeichnis

# II. Abbildungsverzeichnis

# III. Tabellenverzeichnis

# IV. Quellcodeverzeichnis

# V. Abkürzungsverzeichnis

API ..... application programming interface, Seite 62
ARP ..... address resolution protocol, Seite 63
BOS ..... Behörden und Organisationen mit Sicherheitsaufgaben, Seite 22
CMSIS ..... Cortex Microcontroller Software Interface Standard, Seite 69
DFT ..... diskrete Fouriertransformation, Seite 26
DMA ..... direct memory access, Seite 67
DSP ..... Digital Signal Processor, Seite 69
DSP ..... Digitaler Signalprozessor, Seite 40
EMAC ..... Ethernet Media Access Controller, Seite 47
FFT ..... fast fourier transformation/schnelle Fouriertransformation, Seite 26
fifo ..... first in first out, Seite 88
glitch ..... transienter, stochastischer Fehler in Digitalschaltungen, Seite 46
GSM ..... Global System for Mobile Communications, Seite 2
IC ..... Integrated Circuit, Seite 46
ICMP ..... Internet Control Message Protocol, Seite 63
IP ..... internet protocol, Seite 63
ISR ..... Interrupt-Service-Routine, Seite 52
MDI ..... Medium Dependent Interface, Seite 48
MFB ..... multiple feedback topology / Filter mit Mehrfachgegenkopplung, Seite 28
PCM ..... Pulse-Code-Modulation, Seite 2
PHY ..... Baugruppe für Physical Layer, Seite 62
RAM ..... Random Access Memory, Seite 62
ROM ..... Read Only Memory, Seite 62
SNR ..... signal to noise ratio/Signal-Rausch-Abstand, Seite 8
SoC ..... System on Chip, Seite 5
SPI ..... serial peripherial interface, Seite 46
SSP ..... Synchronous Serial Port, Seite 76
TCP/IP ..... Transmission Control Procotol/Internet Prokotoll), Seite 61
UDP ..... User Datagram Procotol, Seite 61
VoIP ..... Voice over IP, Seite 2

# 1 Einleitung

## 1.1 Aufgabenstellung

Im Rahmen dieser Abschlussarbeit soll an der Professur für Kommunikationstechnik der Hochschule Mittweida ein Verfahren zur Synchronisation von Teilnehmern für die Kommunikation über schmalbandige, niederfrequente Analogkanäle entwickelt und testweise implementiert werden. Darüber hinaus sind zwei Geräte zur Sprachübertragung im Richtungsbetrieb über verschiedene Kommunikationskanäle zu entwickeln, wobei eine Verschlüsselung möglich sein soll. Diese ist exemplarisch als Time-Domain-Scrambling-Verfahren, wenn möglich mit einer Blocklänge von 1s und einer Schlüssellänge von mindestens 40 Symbolen, umzusetzen. Die Effektivität und Sicherheit der Verschlüsselung sollen nicht betrachtet werden.
Die Synchronisation ist als Rahmensynchronisation für die Übertragungsblöcke auszuführen. Dazu soll ein Sinusimpuls übertragen werden, welcher vom Empfänger detektiert wird. Das Verfahren zu Erkennung ist frei wählbar. Vorgeschlagen wird die Implementierung des Goertzel-Algorithmus.
Als Hardwarebasis für die Kommunikationsgeräte ist der Mikrocontroller *LPC1768* von *NXP* auf dem Mikrocontrollerboard *Chip1768* mit dem Evaluationsboard *testBed for mBed* von *Elektronikladen* zu verwenden. Entsprechende Schaltungen zur Übertragung und Detektion des Synchronimpulses sind zu entwickeln und zu implementieren, sofern nötig.
Die Firmware des Controllers ist in der Programmiersprache *C* zu entwickeln. Dafür wird die Entwicklungsumgebung $\mu Vision5$ von *KEIL* zur Verfügung gestellt.

## 1.2 Motivation

Ob fachkundig oder nicht, dem geneigten Leser wird bekannt sein, dass sich analoge Verfahren im Bereich der elektronischen Kommunikation auf dem Rückzug befinden und mit voranschreitender Zeit durch flexiblere, stabilere und leistungsfähigere Entwicklungen ersetzt werden. Wieso sollte man dennoch ein Verfahren und die dafür benötigte Hardware zur analogen, verschlüsselten Übertragung von Gesprächen entwickeln?

Zum einen besteht das Interesse, zu überprüfen ob bis dato als veraltet geltende Techniken durch die Verwendung moderner Hard- und Software eine erneute Praxistauglichkeit erreichen könnten, zum anderen besteht die Möglichkeit, dass die Verwendung solcher Verfahren zur Verschlüsselung von Informationen aufgrund ihrer schwindenden Verbreitung eine inhärente Sicherheit gegen großflächig angelegte, automatische Angriffe mit sich bringen. Mit steigender Verbreitung digitaler Übertragungsverfahren und Verschlüsselungsalgorithmen, konzentrieren sich auch Angreifer auf diese Bereiche und

verlieren aufgrund der kleiner werdenden Zielgruppe das Interesse an weniger verbreiteten Umsetzungen. Dies liegt daran, dass die „Rentabilität“ sinkt und der vermeintliche Ruhm in solchen Kreisen mit der Bekanntheit und der Größe des Ziels schwindet. Darüber hinaus stellt sich die Frage ob die Sicherheit solcher Verfahren mit zeitgemäßen Implementierungen soweit erhöht werden kann, dass sie mit heute als Standard geltenden konkurrieren können.

Wie der Aufgabenstellung in Abschnitt 1.1 zu entnehmen ist, werden viele dieser Fragen und Thesen nicht in dieser Arbeit behandelt werden, jedoch sollen damit zumindest die Grundlagen für weiterführende Untersuchungen und Entwicklungen in der Fachgruppe Kommunikationstechnik der Hochschule Mittweida gelegt werden.

## 1.3 Historie und Stand der Technik

Die Übertragung von Informationen auf analogen Kanälen stellt seit Beginn den Grundstein der elektronischen Kommunikation der modernen Zivilisation dar. Die ersten analogen Übertragungen auf elektronischem Wege fanden Anfang des 19. Jahrhunderts in Form der Telegraphie statt. Im Laufe des Jahrhunderts entwickelte sich dieses einfache Verfahren weiter bis zur Telefonie, welche in den 1880er Jahren zu ersten Vermittlungsstellen in Deutschland und den USA führte. Darauf aufbauend entwickelte sich das Tele-Fax mit der die Übertragung von Bildern möglich ist. Parallel dazu entstanden neue, neben den leitungsgebundenen Übertragungstechniken, auch kabellose Funklösungen, deren Fortschritt gegen Ende des 19. Jahrhunderts, zu Beginn vor allem durch die Schifffahrt und später durch militärische Nutzung in den beiden Weltkriegen, vorangetrieben wurde. Vorrangig durch die militärische Forschung, wurde der Einsatz von Verschlüsselungstechniken und somit nötigen Synchronisationsverfahren ergründet. Dabei kamen verschiedene, in Abschnitt 2.7 beschriebene, Verfahren zum Einsatz Die Entwicklung drahtloser Lösungen gipfelte mit der Einführung moderner Mobilfunkverfahren wie GSM in den 90er Jahren des 20. Jahrhunderts.

In den 70er Jahren des 20. Jahrhunderts begann die deutsche Bundespost, damals Betreiber des deutschen Fernsprechnetzes, mit der Digitalisierung ihrer Zwischenamtsübertragungstechnik im Festnetz. Dabei wurden die niederfrequenten Analogsignale bis zur Vermittlungsstelle analog übertragen und von dort, durch *Pulse-Code-Modulation* (*PCM*) digitalisiert, an die Vermittlung des Empfängers übertragen. In der nächsten Ausbaustufe wurden die Vermittlungen digital, sodass letztlich nur noch die Verbindung zwischen Teilnehmer und Vermittlung analog stattfanden. Seit 1989 wurde dieser Abschnitt durch ISDN teilweise modernisiert, die Mehrheit der Anschlüsse blieb jedoch für die nächsten Jahrzehnte analog. Seit einigen Jahren treibt die Deutsche Telekom AG, Nachfolger der deutschen Bundespost, die Digitalisierung ihres Fernsprechnetzes durch die flächendeckende Umstellung auf *Voice over IP* (*VoIP*) voran. [26]

Heute existieren analoge Übertragungsverfahren nur noch in sehr wenigen Bereichen. Dazu zählen vorrangig das Radio, Militärfunk, Amateurfunk, der mobile Seefunkdienst, sowie der Flugfunkdienst. Eine der letzten großen Umstellungen vollzieht sich seit 2010 bei den deutschen Behörden und Organisation mit Sicherheitsaufgaben, welche ihre Funksysteme landesweit von analoger auf digitale Funktechnik umrüsten. [2] Bei Radiodiensten und dem Amateurfunk gibt es seit längerem bestreben diesen Schritt zu vollziehen. Eine flächendeckende Anwendung oder Weiterentwicklung analoger Verfahren ist zukünftig nicht zu erwarten.

Heutige Entwicklungen setzen auf symmetrische und asymmetrische digitale Block- oder Stromchiffren wie AES, RSA oder A5. Diese sind nicht nur verhältnismäßig einfach und schnell in Hard- und Software implementierbar, sie gelten bei richtiger Anwendung auch als sicher. Nichts desto trotz wird weiterhin an der Verbesserung und Entwicklung neuer Algorithmen gearbeitet. Denn leistungsfähigere Hardware ermöglicht nicht nur bessere Verschlüsselungen, in gleichem Maße stehen auch neue Technologien für Angriffe auf diese zur Verfügung.

## 1.4 Aufbau der Arbeit

Nachdem in Kapitel 1 auf die Aufgabenstellung und Hintergründe dieser Arbeit eingegangen wurde, werden in Kapitel 2 die theoretischen Grundlagen für die Entwicklung behandelt. Kapitel 3 befasst sich mit in Frage kommenden Verfahren und der Auswahl sowie Implementierung der favorisierten Technik. Anschließend wird in Kapitel 4 die praktische Realisierung des Laborversuchs dargelegt. Auf die Ergebnisse der Arbeit wird in Kapitel 5 eingegangen. Ein Fazit, sowie ein Ausblick auf weitere Verbesserungen und Entwicklungen, welche auf den Ergebnissen dieser Arbeit basieren sind abschließend in Kapitel 6 zu finden.

# 2 Theoretische Vorbetrachtung

In diesem Kapitel sollen die theoretischen Grundlagen für die digitale Verarbeitung von Audiosignalen, die Synchronisation von Kanälen, sowie mögliche Verfahren für die Verschlüsselungen des Kanals betrachtet werden. Diese sind ausschlaggebend für die Auswahl der verwendeten Techniken, sowie für die Entwicklung der dazu benötigten Hard- und Software.

## 2.1 AD- und DA-Wandlung

Zur Verarbeitung von Informationen mittels Mikrocontrollern, müssen analoge Signale digitalisiert werden. Darunter versteht man die Quantisierung und Diskretisierung des Signals sowohl im Zeit-, als auch im Wertebereich. Dabei sind einige „Grundregeln“ einzuhalten um eine verlustfreie Digitalisierung und Rekonstruktion gewährleisten zu können. Für diesen Vorgang werden diskrete Schaltkreise oder sogenannte „Systems on Chip“ (SoC) verwendet. Bei SoCs handelt es sich um, auf dem Chip des Mikrocontrollers integrierte Wandler, wohingegen diskrete Schaltkreise in eigenen Gehäusen untergebracht werden. Unterschiede bezüglich der Leistungsfähigkeit zwischen diesen beiden Ausführungen gibt es im allgemeinen nicht, allerdings sind SoCs wesentlich kleiner, als eigenständige Schaltkreise. Darüber hinaus, existieren externe Schaltkreise für spezielle Anwendungsfälle, wohingegen SoCs in der Regel für einen breites Anwendungsspektrum ausgelegt werden.

### 2.1.1 Prinzip

Bei der Wandlung von analogen, kontinuierlichen Signalen in digitale, diskrete Werte wird das Signal periodisch abgetastet, wobei zu jedem Abtastzeitpunkt ein repräsentativer Wert ermittelt und digital zur Verfügung gestellt wird. Dabei findet sowohl eine zeitliche Diskretisierung, als auch eine Quantisierung der Werte statt. Ein daraufhin rekonstruiertes analoges Signal entspricht aufgrund von systematischen Fehlern in der Praxis nicht mehr dem Quellensignal. Abbildung 2.1 zeigt die ideale Abtastung (rot) eines analogen Signals (grau).

### 2.1.2 Wandlungsverfahren

Zur Wandlung von analogen in digitale Signale und umgekehrt, haben sich verschiedene Verfahren mit unterschiedlichsten Eigenschaften etabliert. Einige, sollen hier kurz

[1] Quelle: WikiMedia Commons

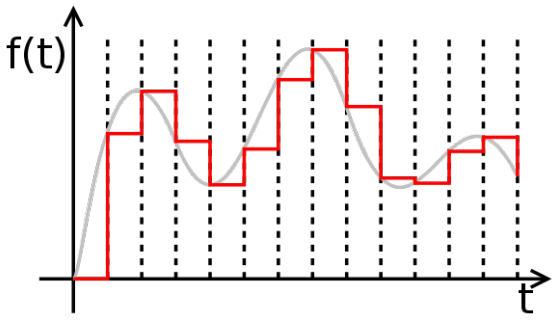



Abbildung 2.1: Abtastung eines kontinuierlichen Signals[1]

vorgestellt werden.

### AD-Wandlung

**Sample and Hold** Zur Digitalisierung von analogen Signalen wird ein sogenanntes „Sample and Hold“-Glied verwendet. Dabei wird der abzutastende Wert in einem Kondensator zwischengespeichert und vom Eingangssignal getrennt. Dies hat zur Folge, dass sich der Eingangswert während des Wandlungsprozesses nicht ändert und der Prozess somit nicht verfälscht wird. Anschließend wird eins der folgend beschriebenen Verfahren zu Ermittlung des digitalen Wertes verwendet.

**Zählverfahren** Bei Verfahren wie z.B. dem **Single-Slope-** oder **Dual-Slope-Verfahren**, wird die Spannung durch das Zählen von Zeiten bzw. Impulsen ermittelt. Dabei wird in beiden Fällen ein Kondensator geladen und die Zeit für die Auf- oder Entladung mit Hilfe eines Oszillators gezählt. Der Zählwert ist ein direktes Maß für die Eingangsgröße. Bei **Single-Slope** wird nur über den Ladevorgang gezählt, wobei der Kondensator mit einer Sägezahnspannung mit konstantem Anstieg geladen und die Momentanspannung mit der Eingangsspannung verglichen wird. Erreicht die Sägezahnspannung die zu konvertierende Spannung, kann die Größe durch den bekannten Anstieg und die vergangene Zeit bestimmt werden. Das **Dual-Slope-Verfahren** hingegen verwendet den Entladevorgang eines Kondensators zur Bestimmung. Dabei wird der Kondensator über die Eingangspannung aufgeladen und anschließend auf ein definiertes Potential entladen, wobei wiederum die Zeit über einen Zähler bestimmt wird. Anschließend steht das Ergebnis auch hier direkt zur Verfügung.

**Serielle Verfahren** basieren in der Regel auf Vergleichsspannungen und Komparatoren. Bei der **sukzessiven Approximation** wird für $n$-Bit in $n$-Schritten eine Vergleichsspannung durch einen Digital-Analog-Wandler erzeugt und mit einem Komparator beginnend bei der größtmöglichen Spannung verglichen. Anschließend wird das entsprechende Bit in einem Register gespeichert. War die Vergleichsspannung größer als der

Eingangswert, so ist der Wert $0$, war sie kleiner ist er $1$. Im darauffolgenden Schritt wird die nächstkleinere Spannung eingestellt. Dieses Verfahren wandelt ein Bit pro Durchlauf. Es ist langsamer als andere Verfahren, jedoch verhältnismäßig einfach umzusetzen.

**Parallele Verfahren** Das **Flash-Verfahren** erfordert verglichen mit seriellen Verfahren eine komplexere Schaltung, ermöglicht jedoch wesentlich kürzere Wandlungszeiten, da der Digitalwert bereits nach einem Schritt zur Verfügung steht. Dies wird dadurch ermöglicht, dass eine abgestufte Referenzspannung mit Komparatoren verglichen und anschließend durch einen Decoder in den digitalen Repräsentanten konvertiert wird.

### DA-Wandlung

Die Digital-Analog-Wandlung dient der Erzeugung analoger Signale aus digitalen Werten. Dabei wird ein, in einem Register gespeicherter Digitalwert in eine korrespondierende, proportionale Analogspannung konvertiert. Dazu stehen unter anderem folgende Verfahren zur Verfügung.

**Direkte Umsetzung** Bei der Direktumsetzung eines Wandlers mit $n$ Bit, wird die Spannung über einen Spannungsteiler aus $2^n$ Widerständen und Schaltern erzeugt. Dabei wird für jede der Stufen der entsprechende Schalter geschlossen, wenn diese durch den Digitalwert gewählt wurde und die Spannung am dazugehörigen Spannungsteiler ausgegeben. Dieses Verfahren ist zwar sehr schnell, erfordert jedoch einen sehr hohen Materialeinsatz und ist praktisch nur für sehr geringe Auflösungen sinnvoll realisierbar.

**R2R-Netzwerk** Ein Vertreter paralleler Verfahren zur DA-Wandlung, ist das R2R-Netzwerk, auch als Wägeverfahren bezeichnet. Dabei handelt es sich um ein Widerstandsnetzwerk welches aus zwei Widerstsandswerten besteht. Eine Kaskadierung entsprechend Abbildung 2.2 ermöglicht eine zur Eingangsbitfolge korrespondierende Analogspannung am Ausgang. Das R2R-Netzwerk mit $n$ Bit Auflösung benötigt mindestens $2 * n$ Widerstände und $n$ Schalter zur Realisierung und erfordert daher einen höheren Aufwand, welcher jedoch in einer sehr kurzen Wandlungszeit resultiert. Zu beachten ist allerdings eine hohe Genauigkeit bei den Widerständen, da Abweichungen einen direkten Einfluss auf das Ergebnis haben.

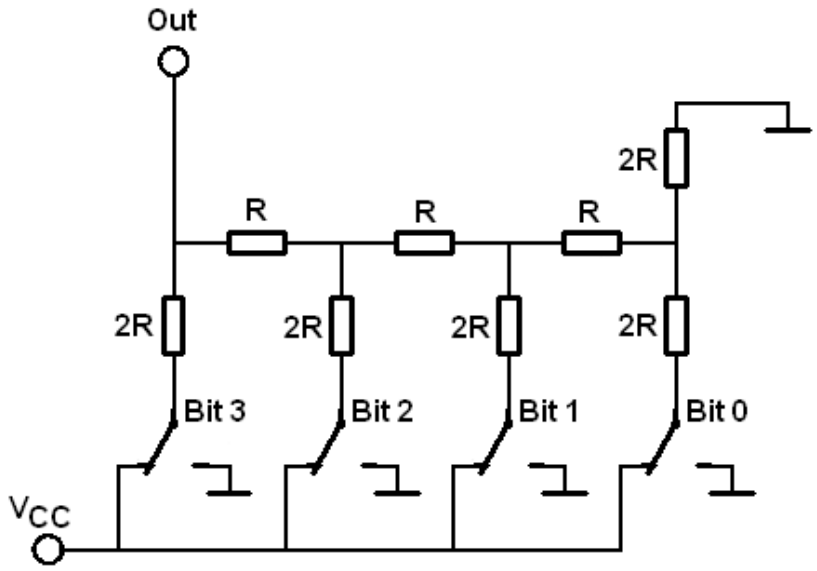



Abbildung 2.2: R2R-Netzwerk als D/A-Wandler

**Zählverfahren** Beim Zählverfahren wird die Ausgangsspannung über das Öffnen und Schließen eines einzigen Schalters realisiert, man spricht deshalb auch von einem *1-Bit Umsetzer*. Über diese Ausgangsimpuls mit definiertem Tastverhältnis wird, häufig mittels eines einfachen Tiefpasses, der Mittelwert gebildet, welcher der ursprünglichen Analogspannung entspricht. Diese Variante findet aufgrund der Gemeinsamkeiten mit dem PWM-Ausgang von Mikrocontrollern häufig Anwendung bei SoC-Lösungen.

**Delta-Sigma-Verfahren** Die Delta-Sigma-Wandlung ähnelt im Grundprinzip dem Zählverfahren. Zusätzlich findet eine Delta-Sigma-Modulation und eine Überabtastung statt. Dies führt zum einen zur Optimierung der spektralen Leistungsdichte des Rauschens und zum anderen zur Verschiebung des Rauschens in höhere, ungenutzte Frequenzbereiche, auch als *noise shaping* bezeichnet. Beim Delta-Sigma-Wandler wird das Analogsignal ebenfalls mit 1 Bit Auflösung erzeugt, durch die Überabtastung erhöht sich diese mit jeder Verdopplung der Abtastfrequenz um 1 Bit, vorausgesetzt wird dabei die Einhaltung des Nyquist-Shannon-Theorems[2]. Das SNR steigt dabei proportional um 9,03dB, wenn es sich dabei um ein Glied 1. Ordnung handelt. [14, S. 318]

### 2.1.3 Signal-Rausch-Abstand (SNR)

Der *Signal-Rausch-Abstand*, auch *signal to noise ratio* (SNR) genannt, ist eine wichtige Bezugs- und Vergleichsgröße in der Informations- und Kommunikationstechnik. Sie ist als das logarithmische Verhältnis in *Bel* zweier mittlerer Leistungen, in der Regel einem Signalpegel und einem dazugehörigem Störpegel, definiert. Das Signal ist dabei

[2] Vgl. Unterabschnitt 2.1.4

die informationstragende Größe, der Störpegel jedes weiteres als Störquelle definiertes Signal, wobei es sich nicht unbedingt nur um Rauschen handeln muss. Es können auch zwei informationstragende Signale sein, wobei jedoch nur eins von Interesse ist und das andere als störend aufgefasst werden kann. Gleichung 2.1 veranschaulicht den SNR eines Nutz- und eines Störsignals, wobei die Rauschleistung die Hälfte der Nutzsignalleistung beträgt. Wie der Gleichung zu entnehmen, entspricht eine Verdopplung der Leistung einem logarithmischen Wert von 3dB.

$$\begin{aligned} SNR &= 10 \cdot \lg \frac{P_{Nutzsignal}}{P_{Störsignal}} \\ &= 10 \cdot lg \frac{2W}{1W} \\ &= \underline{\underline{3.0103dB}} \end{aligned} \tag{2.1}$$

### 2.1.4 Aliasing

Um das abgetastete Signal vollständig rekonstruieren zu können, muss die Abtastfrequenz gemäß dem Nyquist-Shannon-Theorem das doppelte der oberen Grenzfrequenz des Signals betragen. Durch die Abtastung, werden im Spektrum des Signales Wiederholungen der Information im Abstand der Abtastfrequenz erzeugt, sodass hohe und niedrige Frequenzen bei Unterschreiten dieser Grenze überlagert und bei der Rekonstruktion nicht getrennt werden können. Diesen Effekt nennt man **Aliasing**. Je größer der Abstand zwischen oberer Grenzfrequenz und Abtastfrequenz ist, desto größer ist dieser zwischen den spektralen Wiederholungen und desto geringer sind die Anforderungen an den Rekonstruktionsfilter. Abbildung 2.3 veranschaulicht dies am Beispiel eines abgetasteten Leistungsdichtespektrums. Dargestellt sind die Spektren zweier Signale, welche mit jeweils $8\mathrm{kHz}$ und $16\mathrm{kHz}$ abgetastet wurden. Zu sehen ist, dass der Abstand zwischen dem Basisband und der Wiederholung im Spektrum mit größer werdender Abtastfrequenz zunimmt, wenn die Grenzfrequenzen bzw. die Bandbreiten der Signale unverändert bleiben. Darüber hinaus zeigt sich, dass sich die Bänder des blau dargestellten Signales zu überlagern beginnen, wenn die Abtastfrequenz noch geringer werden würde.

### 2.1.5 Rekonstruktion

Um aus dem abgetasteten Signal, welches nach der Abtastung höhere Frequenzanteile enthält, das ursprüngliche Signal zu erhalten, verwendet man Rekonstruktionsfilter. Deren Funktion ist es, alle, durch die Abtastung hinzugekommenen Frequenzanteile, herauszufiltern und somit das ursprüngliche Signal zu rekonstruieren. Man verwendet hierfür auch den Begriff **Antialiasing**. Im theoretischen Modell, kann ein mit doppelter Grenzfrequenz abgetastetes Signal vollständig durch die Filterung mit einem idealen

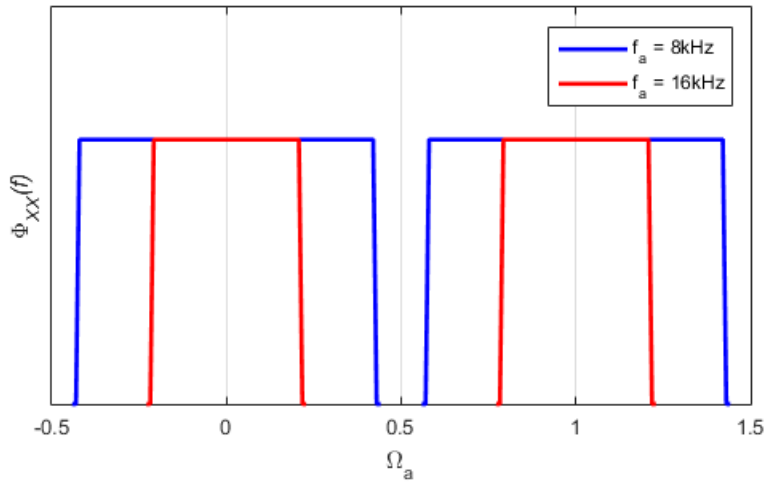



Abbildung 2.3: Veranschaulichung des Aliasing-Effekts im Bildbereich

Tiefpassfilter der Bandbreite $2f_g$ rekonstruiert werden. In der Praxis besitzen Frequenzfilter jedoch nichtideale Eigenschaften, die dazu führen, dass der Anstieg und Abfall der Dämpfung an den Grenzfrequenzen nicht unendlich und somit nicht ideal sind. Um die Qualität der Filterung dennoch erhöhen zu können benötigt man entweder Filter höherer Ordnung um die Dämpfung zu erhöhen, oder man verwendet Verfahren wie Oversampling (Unterabschnitt 2.1.6), um den spektralen Abstand der Bänder zu erhöhen. In der Praxis greift man aus Kostengründen und der immer weiter voranschreitenden Miniaturisierung auf Oversampling zurück. Moderne Hardware ist häufig in der Lage ein vielfaches der Eingangsfrequenz bereitzustellen, sodass auf eine Implementierung von aufwändigeren Analogfiltern verzichtet werden kann.

### 2.1.6 Oversampling

Oversampling, oder zu deutsch Überabtastung, ist eine mittlerweile sehr einfach gewordene Möglichkeit den Aliasing-Effekt und die Komplexität der Rekonstruktionsfilter zu verringern. Dabei wird die Abtastrate, häufig um ihr eigenes vielfaches, und so der spektrale Abstand zwischen den Wiederholungen im Frequenzband erhöht. Dadurch kann der Verstärkungsabfall und somit die Ordnung des Filters bei gleichem Signal-Rausch-Abstand geringer ausfallen oder der SNR bei gleicher Filterordnung erhöht werden. In Abbildung 2.3 ist die Abhängigkeit zwischen spektralem Abstand und Abtastfrequenz deutlich zu erkennen. Abbildung 2.4 zeigt zwei verschiedene Simulationen, in denen ein abgetastetes Signal rekonstruiert werden soll. Dies geschieht mit einem Tiefpassfilter zweiter Ordnung, wobei die Abtastfrequenz einmal bei $8\mathrm{kHz}$ und im anderen Fall bei $16\mathrm{kHz}$ liegt. Zu sehen sind die deutlichen Unterschiede beim Verlauf am Filterausgang. Während in der linken Darstellung deutliche Unterschiede zwischen Eingangs- (blau) und Ausgangssignal (rot) zu sehen sind und der ursprüngliche Verlauf nur zu erahnen ist, ist im rechts dargestellten Versuch eine deutliche Verbesserung zu sehen. Der ab-

getastete, quantisierte Verlauf ist in beiden Diagrammen grün dargestellt.

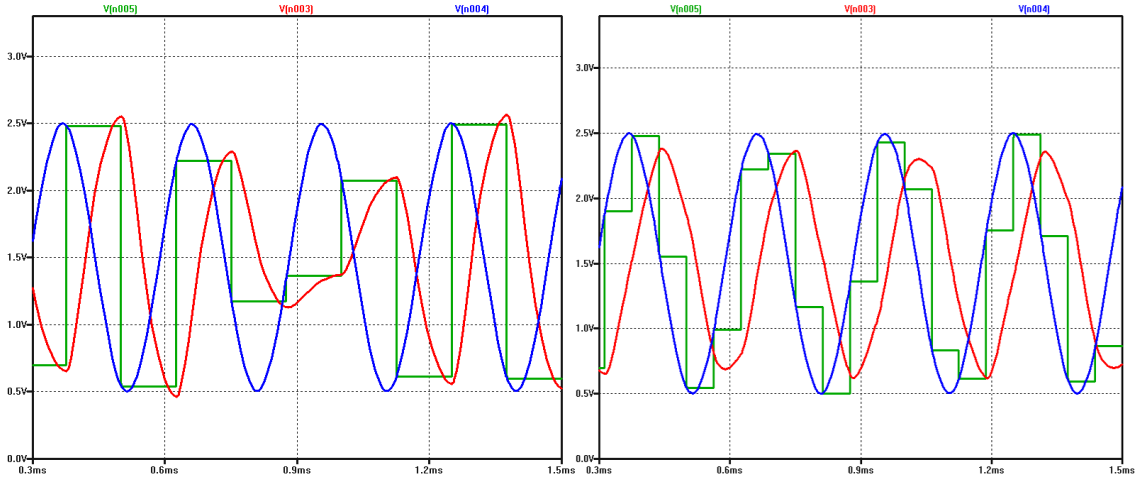



Abbildung 2.4: Auswirkungen der Überabtastung bei der Digitalisierung

Die Optimierung erfolgt dabei jedoch nicht alleine durch eine bessere Filterwirkung, sondern auch durch die Reduktion der Rauschleistung. Unter der Annahme, dass das Quantisierungsrauschen durch bandbegrenztes, mittelwertfreies weißes Rauschen modelliert werden kann, ergibt sich nach der Quantisierung mit dem Fehler $e_q$ nach Gleichung 2.2 eine Rauschleistung $e_{rms}^2$ nach Gleichung 2.3. Der Quantisierungsfehler ergibt sich dabei aus der Hälfte einer Quantisierungsstufe, da dies die mittlere Abweichung darstellt.

$$e_q = \frac{\Delta}{2} = \frac{V_{ref}}{2 \cdot 2^N} \tag{2.2}$$

$$e_{rms}^2 = \int_{-\frac{\Delta}{2}}^{\frac{\Delta}{2}} \frac{e_q^2}{\Delta} \mathrm{d}e = \frac{\Delta^2}{12} \tag{2.3}$$

Die frequenzabhängige Energiedichte des in-band Rauschens ist mit Gleichung 2.4 definiert und über den gesamten Bandabschnitt konstant, da es sich um weißes Rauschen handelt. [23, S. 6] Daraus ergibt sich das Leistungsdichtespektrum $e_{rms}^2(f)$ durch Gleichung 2.5.

$$E(f) = e_{rms} \cdot \left(\frac{2}{f_a}\right)^{\frac{1}{2}} \tag{2.4}$$

$$e_{rms}(f)^2 = E(f)^2 = e_{rms}^2 \cdot \frac{2}{f_a} \tag{2.5}$$

Die Rauschleistung $n_0^2$ berechnet sich allgemein durch Integration der spektralen Leistung über den entsprechenden Frequenzbereich (Gleichung 2.6). Durch Erhöhung der Abtastrate verteilt sich die Rauschleistung proportional zur Oversamplingrate *(OSR)* im Spektrum der Abtastfrequenz, wodurch die Leistung im Basisband des Signals verringert wird (Gleichung 2.7). Die Oversamplingrate, auch Oversamplingfaktor genannt, ist der Quotient aus der Nyquistfrequenz $f_n$ und der tatsächlichen Abtastrate $f_a$.

$$n_0^2 = \int_{f_u}^{f_o} e_{rms}^2(f)\mathrm{d}f \tag{2.6}$$

$$\begin{aligned} n_{0,\mathrm{OSR}}^2 = \int_0^{f_m} e_{rms}^2(f)\mathrm{d}f \quad &= \frac{2}{f_a}\int_0^{f_m} e_{rms}^2\mathrm{d}f \\ = 2e_{rms}^2 \frac{f}{f_a}\Big|_0^{f_m} \quad &= e_{rms}^2\frac{f_n}{f_a} \\ = \underline{\underline{\frac{e_{rms}^2}{\mathrm{OSR}}}} \end{aligned} \tag{2.7}$$

Setzt man die Leistungen ins logarithmische Verhältnis ergibt sich der Term von Gleichung 2.8.

$$\Delta SNR = 10 \cdot \log(\frac{n_0^2}{n_{0,\mathrm{OSR}}^2}) \tag{2.8}$$

Nach Einsetzen der Gleichungen 2.2 bis 2.7, erhält man nach entsprechenden Umformungen Gleichung 2.9. Aus diesem Ausdruck wird ersichtlich, dass sich das SNR logarithmisch mit der Oversamplingrate ändert und somit jede Verdopplung das Rauschen um 3 dB verringert.

$$\Delta SNR = 10 \cdot \log(\mathrm{OSR}) \tag{2.9}$$

Analog dazu erhöht sich die effektive Auflösung des Wandlungsprozesses mit jeder Verdopplung um 0.5 Bit. Wird Gleichung 2.3 in Gleichung 2.7 eingesetzt und nach der Anzahl der Bits $N$ umgestellt, werden diese Abhängigkeit ersichtlich (Gleichung 2.10).

$$N = \log_2 V_{ref} - \frac{1}{2}\log_2 n_0^2 - \frac{1}{2}\log_2 12 - \frac{1}{2}\log_2 OSR \tag{2.10}$$

## 2.2 Analoge Filter

Bei analogen Filtern handelt es sich um klassische, diskrete oder integrierte Strukturen, die von analogen Signalen durchlaufen werden. Dabei findet keine digitale Bearbeitung statt, viel mehr sind es in der einfachsten, passiven Form frequenzabhängige Spannungsteiler, oder bei aktiven Filtern frequenzabhängige Verstärker. Diese Abhängigkeit kann durch kapazitive oder induktive Elemente erzeugt und durch eine Kaskadierung mehrerer Stufen verstärkt werden, wobei die Komplexität und der Schaltungsaufwand zunehmen und ein Abgleich der einzelnen Stufen schwieriger wird. Analoge Filter können als Tiefpass, Bandpass, Bandsperre oder Hochpass entworfen werden und jeweils verschiedene Eigenschaften, sogenannte Filtercharakteristiken besitzen. Diese bestimmten die Filterkoeffizienten und somit die Eigenschaften im Durchlass- und Sperrbereich des Filters. Eine Beispielimplementierung für einen analogen Filter ist in Unterabschnitt 3.3.3 zu finden.
Die Filterwirkung ist abhängig von der Anzahl der energiespeichernden Bauelemente und wird als Ordnung bezeichnet. Mit jeder Ordnung nimmt die Dämpfung um $20\mathrm{dB}/\mathrm{Dekade}$ zu. Demnach besitzt ein Filter 2. Ordnung zwei Blindelemente und dämpft das Signal mit $40\mathrm{dB}/\mathrm{Dekade}$. Die Frequenz bis zu der oder ab der die Wirkung des Filters greift, nennt sich Eck-, Grenz- oder 3dB-Frequenz und bezeichnet den Punkt, an dem der Ausgangspegel gegenüber dem Eingang um 3dB abgefallen ist (Abbildung 2.5). Das Verhältnis von Eingangs- zu Ausgangsgröße ist die Übertragungsfunktion $H(j\omega)$, in Gleichung 2.11 beispielhaft für einen Tiefpass 1. Ordnung hergeleitet. Die Schaltung gestaltet sich nach Abbildung 2.6.

$$\begin{aligned}
\mathscr{H}(j\omega) = \frac{U_a}{U_e} &= \frac{X_C}{X_C + R} \\
&= \frac{\frac{1}{j\omega C}}{\frac{1}{j\omega C} + R} &&= \frac{1}{j\omega C} \cdot \frac{1}{\frac{1}{j\omega C} + R} \\
&= \underline{\underline{\frac{1}{1 + j\omega RC}}}
\end{aligned} \tag{2.11}$$

Durch Betragsbildung erhält man einen reellen Ausdruck, welcher eine Darstellung im Bodediagramm wie in Abbildung 2.5 ermöglicht (Gleichung 2.12).

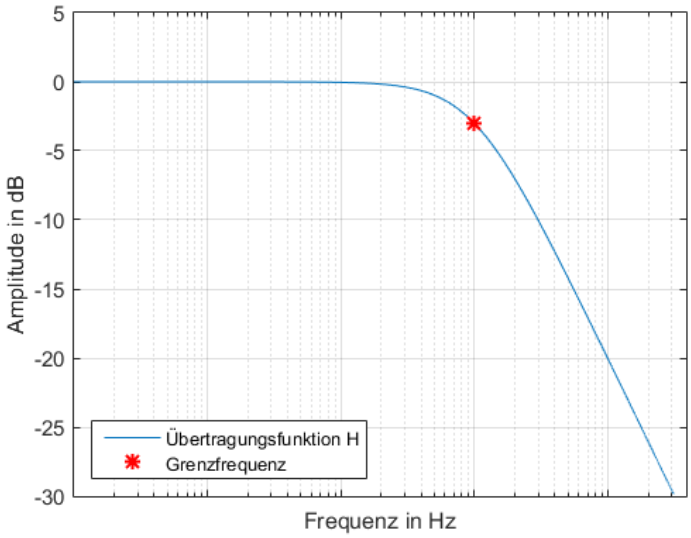



Abbildung 2.5: Bodediagramm eines Tiefpass 1. Ordnung

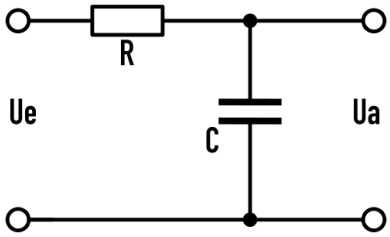



Abbildung 2.6: Schaltung eines Tiefpass 1. Ordnung

$$
\begin{aligned}
|\mathscr{H}(j\omega)| &= \left| \frac{1}{1+j\omega RC} \right| \\
&= \frac{\sqrt{1^2}}{\sqrt{1^2+(j\omega RC)^2}} \\
&= \underline{\underline{\frac{1}{1+(\omega RC)^2}}}
\end{aligned}
\tag{2.12}
$$

## 2.3 Digitale Filter

Digitale Filter sind die moderne Alternative zu analogen Filtern. Dank ausreichend verfügbarer Rechenleistung moderner Controller können sie heute in vielen Anwendungsbereichen realisiert werden. Es handelt sich dabei um eine reine digitale Bearbeitung von Signalen durch mathematische Operationen mit Hilfe von Logikbausteinen oder Software. Sie erfreuen sich großer Beliebtheit, da wesentlich höhere Filterordnungen mit größerer Genauigkeit, weniger Aufwand und geringerem Materialeinsatz realisiert

werden können, solange ausreichend Rechenleistung zur Verfügung steht. Ein weiterer großer Vorteil ist die fehlende Bauteilstreuung, welche bei Filtern höherer Ordnung eine aufwendige und teure Selektion der einzelnen Bauelemente erfordert. Die digitalen Filter gruppieren sich in zwei grundlegende Bereiche, welche sich nach ihrer Struktur und dadurch der Länge der Impulsantwort gliedern. Zum einen gibt es grundsätzlich stabile Filter mit endlicher Impulsantwort, sogenannte FIR-Filter (*finite impulse response*), zum anderen gibt es solche mit nicht endlicher Impulsantwort welche analog als IIR-Filter (*infinite impulse response*) bezeichnet werden und nicht zwingend stabil sind. Bei IIR-Filtern handelt es sich daher im Gegensatz zu FIR-Filtern um rekursive Systeme, bei denen das Ausgangssignal auf den Eingang zurückgeführt wird. IIR-Filter erlauben es jedoch höhere Ordnungen mit geringerer Komplexität zu erzeugen. Allen Filtern gemein ist, dass sie lediglich aus Verzögerungsgliedern, Multiplizierern und Addierern bestehen.
In Unterabschnitt 3.3.4 wird der Entwurf eines digitalen Filters anhand eines Beispiels erläutert.

## 2.4 Synchronisation

Unter Synchronisation versteht man im allgemeinen einen zeitlichen Abgleich zweier Systeme unter Einhaltung einer Toleranz. Bei elektronischen Systemen spricht man von Synchronisation wenn zwei Systeme, z.B. ein Sender und Empfänger eine Taktfrequenz besitzen und ein Gleichlauf nötig ist. Die Synchronisation muss in diesem Fall stattfinden um die Phasenbeziehung der Systeme zu erhalten. Ohne diese würde der Empfänger die Information anders decodieren, als sie vom Sender kodiert wurde.
In den letzten Jahrzehnten haben sich verschiedene Verfahren für die Synchronisation von elektronischen Geräten entwickelt. Man unterscheidet diese nach ihrem Anwendungsfall in synchron, asynchron und plesiochron.

### 2.4.1 Synchroner Betrieb

Synchrone Übertragungsverfahren benötigen während der kompletten Dauer einen gemeinsamen Takt zwischen den Teilnehmern, welcher den zeitlichen Rahmen für die Übertragung definiert. Dadurch wird unter Inkaufnahme einer höheren Komplexität der Systeme eine höhere Datenrate im Vergleich zu asynchronen Übertragungen erzielt. Beide System arbeiten in der Regel mit gleichem Grundtakt. Die Synchronität der Teilnehmer wird über den gesamten Zeitraum der Übertragung aufrechterhalten. [27, S. 150]

### 2.4.2 Asynchroner Betrieb

Bei Asynchronen Übertragungen wird jedes Symbol einzeln Übertragen, nachdem beide Systeme synchronisiert wurden. Es besteht kein direkter zeitlicher Zusammenhang zwischen den Systemen, wodurch zwischen den Symbolen Pausen auftreten können. Asynchrone Verfahren werden deshalb auch als Start-Stopp-Verfahren bezeichnet. Eine Synchronität wird nur über die Dauer eines Übertragungssymbols gewährleistet. Für jedes weitere Symbol, muss eine neue Synchronisierung stattfinden. [27, S. 150] Eine Unterform dieser Betriebsart stellt die isochrone Übertragung dar. Diese findet zwar grundsätzlich asynchron statt, allerdings wird nach jeder Synchronisation eine feste Anzahl von schritten abgearbeitet, sodass jeder Rahmen synchron ist.

### 2.4.3 Plesiochroner Betrieb

Ein relativ selten verwendeter Begriff für ein allerdings häufiger vorkommendes Verfahren ist „plesiochron“. Dabei handelt es sich um eine Mischung der beiden bereits erläuterten Betriebsarten. [24, S. 419] Es findet zwar für jede Übertragung eine Synchronisierung in einem festen Zeitrahmen statt, minimale Abweichungen sind jedoch zulässig und werden durch Stopfbits ausgeglichen, um einen Bitverlust (*Bitslip*) zu verhindern. Darüber hinaus arbeiten die beiden Systeme wie synchrone Teilnehmer mit gleichem Grundtakt. Plesiochrone Verfahren finden zum Beispiel beim Multiplexing von ISDN-Verbindungen Anwendung. Ein großer Nachteil ist der erhebliche Aufwand beim Multiplexen plesiochroner Kanäle, da diese zur Vermittlung bis zur niedrigsten Hierarchieebene demultiplext werden müssen. In großen Knotenpunkten erfordert dies aufgrund der hohen auftretenden Datenraten einen erheblichen technischen Aufwand. [15, S. 452]

## 2.5 Übertragungskanal

In der Praxis existieren einige verschiedene Übertragungskanäle, für welche das entwickelte Verfahren zur Synchronisation Anwendung finden können soll. Darunter sind direkte Leitungsverbindungen, das analoge Festnetz, das Mobilfunknetz, sowie IP-Telefonie (VoIP) und langwelliger Analogfunk. Die Verfahren beeinflussen die Übertragung des Signals auf unterschiedliche Art und Weise, wobei die Synchronisation gegenüber allen Einflüssen stabil bleiben muss. Eine genauere Untersuchung der Übertragungskanäle ist nicht Teil dieser Arbeit, da kein direkter Einfluss auf die Qualität und Störeigenschaften dieser möglich ist. Allerdings wurden abschließend einige Versuche mit verschiedenen Übertragungstechniken durchgeführt um die Funktionalität der Synchronisation zu testen. Allen Übertragungskanälen, außer Voice-Over-IP, gemein ist jedoch die Beschränkung des Frequenzbandes auf einen Bereich zwischen 300 Hz und 3400 Hz. [4, S. 55] Dieses Band wurde ursprünglich für die Übertragung im analogen Tele-

fonnetz belegt. Aus Gründen der Abwärtskompatibilität und der Tatsache, dass dieser Bereich für die meisten Einsatzbereiche eine ausreichend gute Sprachübertragung ermöglicht, wurden die Frequenzen beibehalten. Darüber hinaus wirkt sich eine geringere Bandbreite positiv auf die kumulierte Bandbreite bei analogen Multiplexverfahren aus, weshalb diese aus wirtschaftlichen Gründen so gering wie möglich gehalten wird. [7, S. 1 ff]
Eine Besonderheit bei den Übertragungskanälen bilden die Funkkanäle. Im Gegensatz zu den leitungsgebundenen Übertragungstechniken, bei denen das Signal regelmäßig regeneriert und eine Dämpfung auf dem Kanal nur durch die Impedanz der Leitung und ihrer Verbindungen hervorgerufen wird, werden die Funkwellen durch die Freiraumdämpfung, Abschattung und Reflexionen beeinflusst. Zwar enthalten moderne Systeme wie GSM Verfahren, welche die Leistung des Nutzsignals ermitteln und diese gegebenenfalls anpassen, allerdings muss die Synchronisation auch gegen diese kurzzeitigen Schwankungen der Amplitude im Synchronisationsimpuls tolerant sein und diesen weiterhin zuverlässig detektieren. [7, S. 524]

## 2.6 Fehler und deren Hörbarkeit

Die Qualität der übertragenen Informationen, wird durch die AD- und DA-Wandlung, die eventuelle Verschlüsselung und damit Blockbildung und schwankende Sychronität zwangsläufig verringert werden. Dabei zu beachten ist jedoch die Toleranz des menschlichen Gehörs gegenüber Fehlern bei bekannten, zu erkennenden Mustern, wie Sprache, welche diese Fehlerquellen zumindest teilweise ausgleicht. Entscheidende Kennzahlen des Hörapparates sind dabei die **Mithörschwelle** von Störungen bei der Verdeckung des Nutzsignals, die **Pegelunterschiedsschwelle**, die **Frequenzselektivität** des Gehörs, sowie dessen **Grenzdauern**.

### Mithörschwelle

Kommt es zur bewussten Wahrnehmung eines Audiosignals, zum Beispiel Sprache, welches durch ein Störsignal überlagert wird, tritt der Effekt der Verdeckung auf. Dabei ist es dem Hörer ab einem Bestimmten Pegel nicht mehr möglich die beiden Signale voneinander zu unterscheiden und die Information zu verarbeiten. Damit das Nutzsignal für den Empfänger verständlich wird, muss dessen Pegel erhöht und aus der Störung herausgehoben werden. Dieses Phänomen tritt alltäglich auf, zum Beispiel wenn ein Gespräch durch Umgebungsgeräusche erschwert wird. Mit steigendem Störpegel, muss die Lautstärke der Sprache angehoben werden. Die Mithörschwelle beschreibt dabei den Grenzwert, ab dem ein Nutzsignal bei vorhandenem Störsignal wahrgenommen werden kann.

[3] [1]

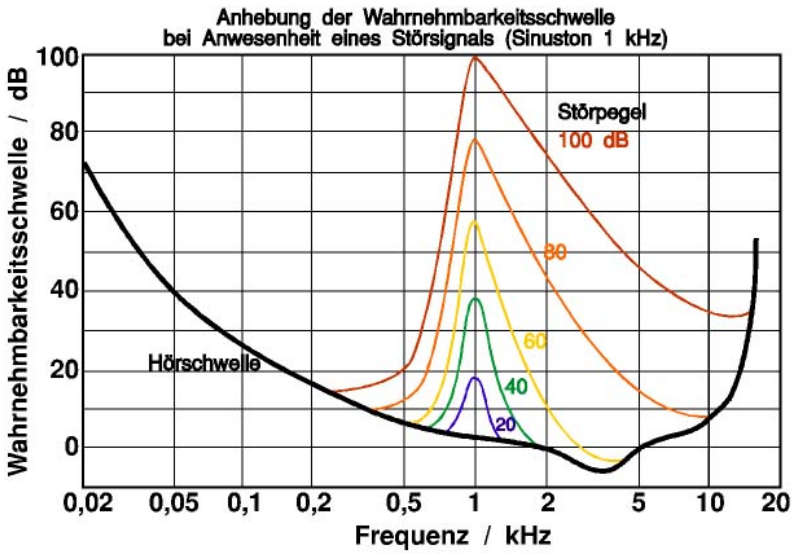



Abbildung 2.7: Ruhehörschwelle und Mithörschwellen des menschlichen Gehörs[3]

Abbildung 2.7 zeigt die Ruhehörschwelle, den Pegel, ab dem das menschliche Gehör ein Geräusch wahrnehmen kann (schwarze Kurve). Zu sehen ist, dass dieser Grenzwert keine Konstante darstellt, sondern äußerst frequenzabhängig ist. Die größte Empfindlichkeit besteht demnach bei 4kHz, dem Frequenzbereich der menschlichen Sprache. Zu höheren und niedrigeren Frequenzen nimmt diese ab. Dieser Effekt wird auch als Frequenzselektivität bezeichnet. Die darüber hinaus dargestellten Mithörschwellen zeigen die Empfindlichkeit des Gehörs bei Anwesenheit eines 1kHz Sinustons. Aus dem Diagramm lässt sich somit ermitteln, dass ein Gespräch oder ein Nutzsignal mit 4kHz einen Pegel von mindestens 60dB haben muss um wahrgenommen zu werden, wenn der Pegel des Störsignals 100dB beträgt. Diese Eigenschaft ist nicht nur von Vorteil für die analoge Übertragung von Audiosignalen, sie wird auch bei der digitalen Verarbeitung von hörbaren Spektren ausgenutzt. So wird bei Codierungsalgorithmen wie MP3 die Verdeckung von Tönen verwendet um irrelevante Informationsanteile herauszufiltern, welche aufgrund der Präsenz anderer Frequenzen nicht oder nur noch schwer wahrnehmbar sind. [28, S. 282]

## Grenzdauer

Da es sich beim menschlichen Gehör um ein elektromechanisches, schwingendes System handelt, ist die Wahrnehmungsfähigkeit in verschiedenen Richtungen beschränkt. Die zeitliche Grenze bis zu der eine Auflösung noch möglich ist, liegt für die in Abbildung 2.7 dargestellten Mithörschwellen, sowie der Ruhehörschwelle zum Beispiel bei ca. 200ms. Ist die Zeitdauer des Nutzsignals geringer, muss dessen Pegel umso höher sein. Das Produkt von Dauer und Intensität bleiben somit gleich. Daraus resultiert, dass das Gehör gegen kurzzeitige Störeinflüsse unempfindlicher ist als gegen lange.

Eine weitere Grenzdauer ist bei 20 ms zu finden. Unter dieser Zeit fällt es schwerer, Schwankungen in Amplitude und Frequenz wahrzunehmen. Die Verständlichkeit wird geringer. Dieser Effekt wird auch bei analogen Verschlüsselungsverfahren ausgenutzt, wobei Teilabschnitte bzw. Sendesymbole des Nutzsignals in so kurze Abschnitte zerlegt und vertauscht werden, dass diese nicht mehr unterschieden werden können. Die unterste Grenzdauer liegt schließlich bei 2 ms. Unter dieser Zeitspanne ist das Gehör nicht mehr in der Lage die Struktur eines Signals auszuwerten, da die Einschwingzeit des Gehörapparates unterschritten wird. [28, S. 290]

## Pegelunterschiedsschwelle

Durch die Nichtlinearität der Verdeckung und der Mithörschwellen, ergeben sich weitere Besonderheiten bei der Wahrnehmung des Pegels von Audiosignalen. Es wurde festgestellt, dass sich die Wahrnehmung von Pegelschwankungen in Frequenz, Lautstärke und Spektrum unterscheidet. In der Nähe der Ruhehörschwelle ist das Gehör so nicht in der Lage Schwankungen mit einer Frequenz von $4\mathrm{Hz}$, dem für Schwankungen empfindlichsten Bereich, und Modulationsgraden bis 0.2 zu unterscheiden. Bis zu einem Pegel von etwa $20\mathrm{dB}$ gilt dies für sämtliche hörbaren Audiosignale. Darüber entwickeln sich die Verläufe für schmalbandige Töne, wie Sinussignale, und breitbandige Geräusche, wie weißes Rauschen, unterschiedlich. Während der Modulationsgrad bei schmalbandigen Signalen mit dem Pegel weiter abnimmt, stabilisiert sich dieser bei breitbandigem Rauschen bei ca. 0.4. Dies hat zur Folge, dass Schwankungen sinusförmiger Signale bei steigender Lautstärke eher wahrgenommen werden als bei Rauschen. Da Sprachübertragungen durch breitbandiges Rauschen angenähert werden können, erlaubt diese Eigenschaft eine minimale Schwankung des Pegels von mindestens 1dB, wohingegen es bei schmalbandigen Signalen weniger als 0,2dB sind. [28, S. 288]

## Verzerrungen von Phasen- und Amplitudenfrequenzgang

Wie bei allen System zur Übertragung von Signalen, ist natürlich auch bei Audiosystem eine möglichst linearer Frequenzgang der Amplitude erwünscht um eine Verfälschung der Informationen durch das System zu verhindern. Wie bereits beschrieben, hat das menschliche Gehör jedoch eine natürliche Toleranz gegenüber solcher Störungen. Aus den Eigenschaften der Pegelunterschiedsschwelle, lässt sich schließen, dass kurzzeitige Einbrüche der Amplitude mit einem Pegel von $1\mathrm{dB}$ nicht hörbar sind. Dies entspricht immerhin einer Änderung von etwa 26%. Da diese Werte jedoch im direkten Vergleich von Tönen ermittelt wurden, und dieser in der Natur so gut wie nie auftritt, können für diese Grenze Werte bis zu 3dB, angenommen werden.
Noch störunempfindlicher zeigt sich das Gehör bei frequenzabhängigen Änderungen der Phase. Diese spielen in der Praxis bis auf wenige Ausnahmen gar keine Rolle, sodass bei der Konstruktion von Audiosystemen keine Rücksicht genommen werden

muss. Ausnahme bilden hier allerdings Kopfhörer, sowie die Anwesenheit leistungsstarker, niederfrequenter Signale, welche ein höherfrequentes Nutzsignal überlagern. [28, S. 292]

## 2.7 Verschlüsselung

Grundlegend unterscheidet man in der Kryptohraphie Verfahren, welche die Gestalt der Information durch Substitution oder durch Transposition verändern und somit unlesbar machen sollen. Beide Arten der Kryptohraphie sind bereits seit vielen tausend Jahren bekannt und werden bis heute, mittlerweile auch kombiniert, eingesetzt.

Historisch bekannte Verfahren sind z.b. die Caesar-Chiffre, eine monoalhphabetische Substitutionschiffre, welche durch den römischen Kaiser Julius Caesar während seiner Feldzüge benutzt worden sein soll, oder die Vigenère-Chiffre, eine polyalphabetische Substitutionschiffre. Beide vertauschen die einzelnen Symbole mit Abbildungen eines Verschlüsselungsalphabets. Dabei kann ein Zeichen des Klartextes ein (monoalphabetisch), oder mehrere (polyalphabetisch) korrespondierende im Geheimalphabet besitzen.

Neben der Substitution ist auch die Transposition hinlänglich bekannt. So wurde die Skytale, das älteste bekannte militärisch genutzte Verschlüsselungsverfahren, bereits vor über 2500 Jahren von den Spartanern eingesetzt. Dabei wird nicht die Gestalt der einzelnen Zeichen einer Information, sondern die Gestalt der Nachricht verändert, indem die Position der Zeichen geändert wird. Nur mit dem entsprechendem Schlüssel, der die Reihenfolge der Vertauschung vorgibt, kann eine Nachricht korrekt entschlüsselt werden. Bei der Skytale repräsentiert ein Holzstab den Schlüssel. Zur Verschlüsselung wird ein Stück Papier um diesen gewickelt und anschließend in Richtung des Stabes beschriftet. Wird das Papier abgerollt, erhält man eine unlesbare Nachricht, da die Reihenfolge der Buchstaben vertauscht wurde. Zur Entschlüsselung wird ein Stab gleichen Durchmessers benötigt (Abbildung 2.8).

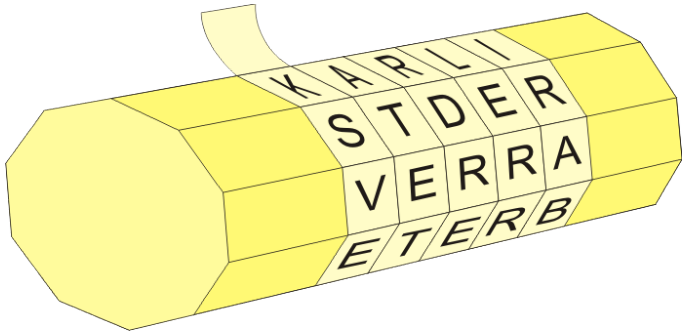



Abbildung 2.8: Symmetrische Transpositionsverschlüsselung - Die Skytale[4]

Beide Arten der Verschlüsselung können für analoge Verschlüsselungen eingesetzt

werden, wobei hier lediglich Verfahren zur Transposition aufgezeigt werden sollen.

### 2.7.1 Verfahren

#### Scrambling [5]

Allgemein versteht man unter Scrambling das Vertauschen von Informationsanteilen. Es gibt zwei wesentliche Verfahren, welche nach der Bezugsgröße unterschieden werden. Sie können sowohl einzeln, als auch in Kombination mit anderen Verfahren angewendet werden. Je nach verwendeter Implementierung, kann eine Frame-Synchronisierung nötig sein.

**Time Domain Scrambling** Beim Time Domain Scrambling (*TDS*) ist die zur Vertauschung herangezogene Bezugsgröße die Zeit. Die Information wird in Blöcke unterteilt, auf welche das Verfahren jeweils getrennt angewendet wird. Anschließend findet eine weitere Unterteilung in Symbole statt, deren Anzahl der Länge des Schlüssels entspricht. Zuletzt werden die einzelnen Symbole entsprechend des Schlüssels vertauscht, wobei jede Ziffer der neuen Position entsprechen könnte. Andere Paradigmen sind dabei allerdings auch denkbar. Das verschlüsselte Signal ist bei ausreichender Blocklänge von einigen hundert Millisekunden unverständlich und kann als sehr sicher gegenüber Angriffen betrachtet werden, sofern eine sinnvolle Schlüsselauswahl erfolgt ist. [18, S. 2]

**Frequency Domain Scrambling** Anders als bei TDS wird bei Frequency Domain Scrambling (*FDS*) mit der Frequenz gearbeitet. Grundlegend sind beide Verfahren jedoch analog. Bei dieser Variante findet die Vertauschung allerdings im Frequenzbereich statt. Dazu wird das Spektrum in eine definierte Anzahl von Subbändern unterteilt, welche anschließend permutiert werden. Auch hier sind verschiedene Paradigmen bezüglich des Schlüssels und seiner Bedeutung denkbar. So können dem Schlüssel beispielsweise verschiedene Positionen in der Reihenfolge der Subbänder zugeordnet werden. Technisch wird FDS häufig mit Filterbänken und Wavelet-Transformatoren realisiert. [12, S. 3] Da die Anzahl der Subbänder und somit die Schlüssellänge begrenzt ist, ergibt sich eine Einschränkung bezüglich der Sicherheit. Hinzu kommt, dass sich aufgrund der jeweiligen Bandcharakteristiken auf das Subband schließen lässt und sich die Anzahl der effektiven Schlüssel auf eine sehr geringe Zahl reduziert. In [18] wird für eine Unterteilung in zum Beispiel fünf Subbänder eine Anzahl von gerade einmal zwölf nutzbaren Schlüsseln festgestellt, was den Aufwand eines Angriffs, z.B. durch

[4] [3]
[5] [12]

Brute-Force, deutlich verringert und das Finden eines Schlüssels innerhalb kürzester Zeit ermöglicht.

### Invertierung [6]

Die Frequenzinvertierung stellt eine einfache Methode zur Verschlüsselung von Audiosignalen im Basisband dar. Dabei wird das Vorzeichen des korrespondierenden Nyquist-Samples invertiert, sodass der frequenzabhängige Verlauf der spektralen Leistungsdichte umgekehrt wird. Anschließend kann optional noch eine Bandverschiebung stattfinden (Abbildung 2.9). Beispielsweise wurde dieses Verfahren vor der Einführung des Digitalfunks bei den deutschen Behörden und Organisationen mit Sicherheitsaufgaben (BOS) eingesetzt. Der alleinige Einsatz einer Frequenzinvertierung muss jedoch als unsicher angesehen werden, da der Inhalt alleine durch wiederholtes Üben erkannt werden kann und die technischen Anforderungen zur Umkehrung des Verfahrens äußerst gering sind. Andere Verfahren sind diesem folglich vorzuziehen. [18, S. 2]

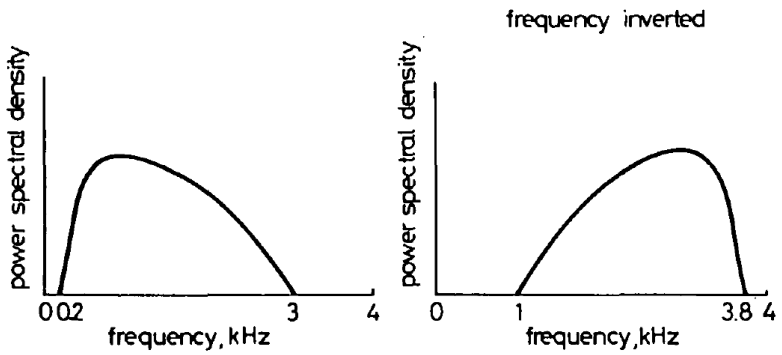



Abbildung 2.9: Spektrale Darstellung der Frequenzinvertierung [7]

[6] [10]
[7] [18]

# 3 Entwicklung des Synchronisationsverfahrens

Dieses Kapitel beschäftigt sich mit der eigentlichen Entwicklung des Synchronisationsverfahrens. Nachdem die zugrunde liegenden Randbedingungen und Voraussetzungen erläutert wurden, werden verschiedene Verfahren, welche zur Detektion des Synchronisationssignals getestet wurden, beschrieben. Darüber hinaus wird auf die Implementierung des letztlich gewählten Verfahrens und der dafür benötigten Peripherie eingegangen.

## 3.1 Randbedingungen

Aus den in Kapitel 2 erörterten Grundlagen und der Aufgabenstellung dieser Abschlussarbeit ergeben sich einige Randbedingungen, welche im Folgenden zusammengefasst werden sollen.

### 3.1.1 Hardwarebasis

Als Hardwarebasis dienen ein 32Bit Miktrocontroller von *NXP* auf dem Entwicklungsmodul *Chip1768* der Firma *Elektronikladen* sowie eine dazugehörige Entwicklungsplatine mit der Bezeichnung *TestBed for mBed*. Bei dem Controller handelt es sich um den LPC1768, welcher in der Cortex-M3 Architektur von ARM aufgebaut ist. Beide Bestandteile werden bereits in der Fachgruppe Kommunikationstechnik der Hochschule Mittweida eingesetzt, weshalb auch diese Entwicklung darauf aufsetzen soll. Eine größtmögliche Kompatibilität zu anderen Entwicklungen mit gleicher Hardware soll dadurch erreicht werden.

### 3.1.2 Software

Die Firmware für den Mikrocontroller ist in C99 zu entwerfen. Als Entwicklungsumgebung kommt eine lizensierte Version von $\mu$Vision 5 des ARM-Distributors *KEIL* zum Einsatz. Mit dieser Umgebung können die verfügbaren Bibliotheken welche dem *Cortex Microcontroller Software Interface Standard* (*CMSIS*) entsprechen verwendet werden. Dazu gehört z.B. die Bibliothek *CMSIS DSP*, welche umfangreiche Funktionen zur digitalen Signalverarbeitung zur Verfügung stellt. Eine unter Windows lauffähige Software mit grafischer Oberfläche zur Konfiguration der Systeme ist mit .NET von Microsoft in C# zu entwickeln.

### 3.1.3 Frequenzbereiche

Die Übertragung soll im Frequenzbereich zwischen 300 Hz und 3400 Hz, dem Basisband der analogen Telefonie, erfolgen[8]. Der Synchronimpuls ist dabei als Sinusfolge mit einer Frequenz von 1600 Hz festzulegen. Der Filter zur Detektion des Impulses soll entsprechend der Möglichkeiten der Hardware so schmalbandig wie möglich sein, wobei ein Kompromiss zwischen der Störanfälligkeit und der sicheren Detektion des Impulses im Basisband eingegangen werden muss.

### 3.1.4 A/D- und D/A-Wandlung

Die A/D-Wandlung erfolgt mit einem externen Wandler, der über SPI mit dem Mikrocontroller zu verbinden ist. Zu Verwenden ist der *MCP3001* von *Microchip*. Der A/D-Wandler besitzt eine Auflösung von 10 Bit, wobei aufgrund des begrenzten Hauptspeichers nur 8 Bit für die Digitalisierung des Nutzsignales zu verwenden sind, und eine mögliche Samplerate von 200 kSamples/sec. Die Ausgabe des Analogsignals mit 8 Bit Auflösung wird durch den Internen Wandler des Mikrokontrollers übernommen.

### 3.1.5 Exemplarische Verschlüsselung

Zur Veranschaulichung und Verifikation der Funktionsweise ist eine Verschlüsselung nach dem TDS-Verfahren zu implementieren, dabei beträgt die Blocklänge $1\mathrm{s}$. Verschiedene Schlüssellängen zur Demonstration der Verständlichkeit sollen wählbar sein. Die Verschlüsselung soll als exemplarisch angesehen werden, und erfüllt keinen Anspruch auf Sicherheit der Übertragung. Bedingt durch die Anwendung des Simplex-Modus ist ein Gerät als Sender zur Verschlüsselung des Nutzsignales und ein weiteres zur Entschlüsselung zu deklarieren.

## 3.2 Systemspezifikation

Entsprechend der Randbedingungen in Abschnitt 3.1 ergeben sich für die Entwicklung des Systems die in Tabelle 3.1 aufgeführten Systemspezifikationen.

Unter Berücksichtigung des verfügbaren Speichers, der Rechenleistung des Controllers und der damit einhergehenden Komplexität des digitalen Filters zur Erkennung des Synchronimpulses, des SNR zur Sicherstellung einer annehmbaren Übertragungsqualität, sowie der nötigen Block- bzw. Symbollänge für die Verschlüsselung wurde eine

[8] Vgl. Abschnitt 2.5

| Eigenschaft | Parameter | Wert | Bemerkung |
|---|---|---|---|
| Basisband | $f_u$ | 300 Hz | |
| | $f_o$ | 3400 Hz | |
| ADC/DAC | Abtastfrequenz | 16 kHz | |
| | Auflösung | 8 Bit | 10 Bit für Synchronimpuls |
| Verschlüsselung | Verfahren | TDS | |
| | Blocklänge | 1s | |
| | Schlüssellängen | 0, 10, 20, 40 | 0 enstpricht Klartext |
| | Symboldauer | 20 ms bis 1 s | abhängig von Schlüssel- und Blocklängen |

Tabelle 3.1: Systemspezifikationen des zu entwickelnden Systems

Abtastfrequenz von $16\mathrm{kHz}$ gewählt. Die in Unterabschnitt 2.1.6 erläuterten und in Abbildung 2.4 dargestellten Simulationen zeigten außerdem, dass die Abtastung mit $16\mathrm{kHz}$ eine ausreichend gute Annäherung an das Ursprungssignal ermöglicht.

## 3.3 Verfahren zur Synchronisation

Zur Erkennung des Synchronimpulses stehen verschiedene Verfahren zur Verfügung, welche im Vorfeld der Entwicklung auf ihre Tauglichkeit untersucht wurden. Bei der Auswahl des Verfahrens spielten vor allem der Berechnungsaufwand und die Komplexität eine wichtige Rolle.

### 3.3.1 Fourier-Transformation

Die Fourier-Transformation ist eine mathematische Operation zur Transformation von Funktionen vom Zeitbereich in den sogenannten Bild- oder Frequenzbereich. Dabei entsteht eine Abbildung des Spektrums, wobei der Beitrag zum Gesamtsignal für jeden Frequenzanteil bestimmt werden kann. Die Fourier-Transformation bietet dadurch die Möglichkeit ein Signal auf die Präsenz einer bestimmten Frequenz hin zu untersuchen. Die Transformation $\mathscr{F}(\omega)$ der Funktion $f(x)$ ist gemäß Gleichung Gleichung 3.1 definiert.

$$\mathscr{X}(f) = \mathscr{F}\{x(t)\} = \int_{-\infty}^{+\infty} f(x) \cdot e^{-\mathrm{j}\omega t}\,\mathrm{d}t \qquad \omega = 2\pi f t \tag{3.1}$$

Führt man diese Transformation zum Beispiel für eine kontinuierliche Cosinusschwigung mit der Frequenz 100Hz durch, erhält man die Spektralfunktion nach Gleichung 3.2 aus der ersichtlich wird, dass die Funktion aus zwei Diracs bei $\pm 100Hz$ mit der Höhe $\frac{1}{2}$ besteht. Die Fouriertransformierte des Cosinus besitzt demnach eine kausale, harmonische Schwingung bei 100Hz.

$$\begin{aligned} \mathscr{X}(f) &= \int_{-\infty}^{+\infty} cos(2\pi \cdot 100Hz \cdot t) e^{-\mathrm{j}\omega t} \mathrm{d}t \\ &= \int_{-\infty}^{+\infty} \frac{e^{\mathrm{j}2\pi 100Hz \cdot t} + e^{-\mathrm{j}2\pi 100Hz \cdot t}}{2} dt \\ &\vdots \\ \mathscr{X}(f) &= \frac{1}{2}(\delta(f+100Hz) + \delta(f-100Hz)) \end{aligned} \tag{3.2}$$

Durch den Übergang zu einem diskretem Definitionsbereich ergibt sich die diskrete Fouriertransformation (*DFT*) nach Gleichung 3.3, welche durch den Algorithmus der schnellen Fouriertransformation (*FFT*) implementiert werden kann. Die DFT bietet den Vorteil, dass mit ihr das komplette Spektrum eines zu untersuchenden Signales berechnet werden kann. Durch die große Anzahl von Rechenoperationen, eignet sich der Einsatz allerdings nicht für jede Anwendung.

$$\mathscr{X}[k] = \mathscr{F}_*\{x[k]\} = \sum_{k=-\infty}^{+\infty} x[k] \cdot e^{-\mathrm{j}\Omega k} \qquad \Omega = 2\pi f T_A \tag{3.3}$$

Eine testweise Implementierung der FFT mit Hilfe der *CMSIS-DSP* Bibliothek von *ARM* zeigte, dass alleine die Ausführungszeit der Transformation mit einer entsprechenden Frequenzauflösung um ein vielfaches über der Abtastperiodendauer von $62.5\mu s$ lag. Mit einer gewünschten Auflösung $A$ von $125\mathrm{Hz}$ und einer Abtastrate $f_s$ von $16kHz$ ergibt sich ein zu transformierender Vektor der Länge $N = 128$ (Gleichung 3.4). Für die Berechnung einer Transformation dieser Länge benötigt der LPC1768 mit einer Taktfrequenz von $100MHz$ ca. $142\mu s$. Dieser Ansatz wurde für diese Anwendung aus diesem Grund nicht weiter verfolgt.

$$N = \frac{f_s}{A} \tag{3.4}$$

### 3.3.2 Goertzel-Algorithmus

Im Gegensatz zur FFT berechnet der Goertzel-Algorithmus Amplitude und Phase einer einzelnen vorgegebenen Frequenz und nicht das Spektrum in Gänze, nutzt dabei jedoch ebenfalls die Eigenschaften der Fourier-Transformation. Dies hat bei einer oder einigen wenigen interessanten Frequenzen einen hohen Geschwindigkeitsvorteil. Der Goertzel-Algorithmus wird z.B. bei der Erkennung von Tonwahlverfahren in der Telekommunikation eingesetzt, da er auch auf einfachen Controllern mit verhältnismäßig geringer Taktrate implementiert werden kann.

Die Systemfunktion des Goertzel-Filters im Bildbereich stellt sich nach Gleichung 3.5 dar. Aus dieser lässt sich ableiten, dass es sich um einen einfach rückgekoppelten, rekursiven IIR-Filter 2. Ordnung handelt. Die einzelnen Filterstufen können mit den z-Transformierten Gleichung 3.6 und Gleichung 3.7 beschrieben werden. Multipliziert man diese ergibt sich die Übertragungsgleichung $H(z)$ (Gleichung 3.8). Wie in der Darstellung zu erkennen, besitzt die Funktion ihre Polstellen bei $e^{\pm j\Omega}$, womit das System als grenzstabil anzusehen ist. Durch Rundungsfehler der notwendigen Festkommaimplementierung besteht somit die Gefahr, dass der Filter instabil wird und das Ergebnis verfälscht. Darüber hinaus erfolgt die Berechnung der Amplituden ebenso wie bei der FFT nur Blockweise, wobei die Blockgröße entscheidend für die Frequenzauflösung ist. Daraus ergab sich nach einem Versuch mit dem LPC1768, analog zur FFT, dass die Berechnung mehr Zeit als vorhanden in Anspruch nimmt, sofern die gewünschte Auflösung für eine entsprechend geringe Bandbreite erreicht werden soll. Auch dieser Algorithmus scheidet somit als Lösungsansatz zur Echtzeiterkennung der Synchronimpulses aus.

$$y[k] = x[k] + e^{-\mathrm{j}\Omega} y[k-1] \tag{3.5}$$

$$\begin{aligned} H_1(z) &= \frac{1}{1 - 2\cos(\Omega) z^{-1} + z^{-2}} \\ &= \frac{1}{(1 - e^{-\mathrm{j}\Omega} z^{-1})(1 - e^{\mathrm{j}\Omega} z^{-1})} \end{aligned} \tag{3.6}$$

$$H_2(z) = 1 - e^{-\mathrm{j}\Omega} z^{-1} \tag{3.7}$$

$$\begin{aligned} H(z) = H_1(z) \cdot H_2(z) \quad &= \frac{1 - e^{-\mathrm{j}\Omega} z^{-1}}{(1 - e^{-\mathrm{j}\Omega} z^{-1})(1 - e^{\mathrm{j}\Omega} z^{-1})} \\ H(z) \quad &= \frac{1}{1 + e^{-\mathrm{j}\Omega} z^{-1}} \end{aligned} \tag{3.8}$$

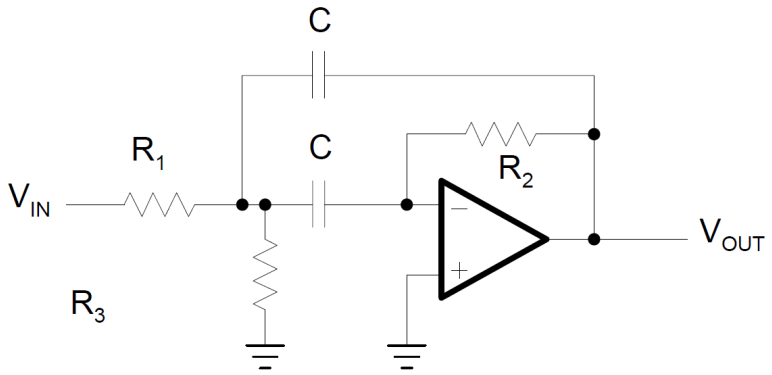



Abbildung 3.1: Aktiver Bandpass mit *multiple feedback topology*[9]

### 3.3.3 Analogfilter

Neben den softwarebasierten Ansätzen, welche in den beiden vorigen Abschnitten beschrieben worden sind, wurde ein diskreter Aufbau bestehend aus einem kaskadiertem zweistufigem Analogfilter 2. Ordnung realisiert. Vorteil des Analogfilters ist die nicht vorhandene Laufzeit der Software. Zwar besitzt auch dieser eine Verzögerung durch die Einschwingzeit, welche durch die kapazitiven Elemente bedingt sind, allerdings sind keine Grenzen durch die Rechenleistung des Controllers gesetzt. Das Signal kann nach dem Filter mit diskreten Logikschaltkreisen aufgearbeitet und später als Digitalsignal am Eingang des Mikrocontrollers ausgewertet werden.
Für die Implementierung eines diskreten Analogfilters wurde ein aktiver Bandpass zweiter Ordnung mit Multiple-Feedback-Topologie (*MFB*) gewählt (Abbildung 3.1). Dieser ermöglicht eine hohe Güte und dämpft das Signal aufgrund der Ordnung mit 40dB/Dekade. Als kaskadierter zweistufiger Filter verdoppelt sich diese. Allgemein lassen sich solche Filter mit Gleichung 3.9 beschreiben.

$$H(s) = \frac{-k \cdot a_1 s}{s^2 + a_1 s + a_0} \tag{3.9}$$

Durch Anwendung des Knotensatzes nach Kirchhoff ergibt sich für die Schaltung in Abbildung 3.1 die Systemgleichung 3.10

$$\begin{aligned} H(s) &= \frac{-\frac{R_2}{2R_1} \cdot \frac{2}{R_2 C^2} s}{s^2 + \frac{2}{R_2 C} s + \frac{1}{R_2 C^2}\left(\frac{1}{R_1} + \frac{1}{R_3}\right)} \\ &= \frac{-\frac{s}{R_1 C}}{s^2 + \frac{2}{R_2 C} s + \frac{1}{R_2 C^2}\left(\frac{1}{R_1} + \frac{1}{R_3}\right)} \end{aligned} \tag{3.10}$$

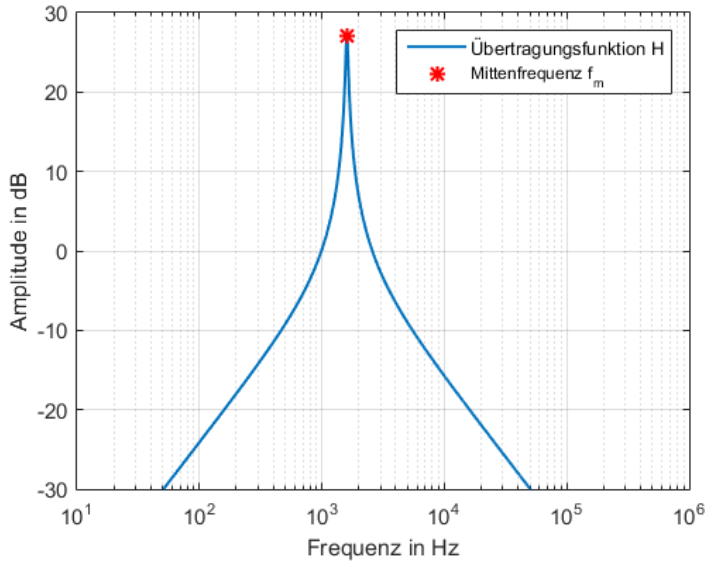



Abbildung 3.2: Bodediagramm des MFP-Bandpasses

Nach Durchführung eines Koeffizientenvergleiches mit Gleichung 3.9 erhält man die Filterkoeffizienten $a_0$, $a_1$ und $k$ (Gleichung 3.11).

$$\begin{aligned} a_0 &= \frac{1}{R_2C^2}\left(\frac{1}{R_1}+\frac{1}{R_3}\right) \\ a_1 &= \frac{2}{R_2C} \\ k &= \frac{R_2}{2R_1} \end{aligned} \tag{3.11}$$

Überführt man die Übertragungsfunktion (Gleichung 3.12) in ihre Fourierdarstellung, ergibt sich der Frequenzgang des Filters, welcher in Abbildung 3.2 dargestellt ist.

$$H(\mathrm{j}\omega) = \frac{-\frac{\mathrm{j}\omega}{R_1C}}{-\omega^2 + \frac{2\mathrm{j}\omega}{R_2C} + \frac{1}{R_2C^2}\left(\frac{1}{R_1}+\frac{1}{R_3}\right)} \tag{3.12}$$

Aufgrund der hohen Verstärkung der Mittenfrequenz $f_m$ lässt sich das Synchronisationssignal sehr gut detektieren, da eine hohe Selektivität zu anderen Frequenzen besteht. Am Filterausgang entsteht ein Sinussignal, sobald der Synchronimpuls übertragen wird. mit Hilfe eines Schmitttriggers wird das Signal anschließend in einen Rechteckimpuls umgewandelt, welcher wiederum über einen Digitaleingang des Controllers ausgewertet werden kann.

[9] [25, S. 16-18]

Nach Implementierung konnte festgestellt werden, dass diese Methode ausreichend genau ist um die Teilnehmer zu synchronisieren, allerdings muss diese für sehr kleine zu detektierende Bandbreiten dennoch als nicht praxistauglich angesehen werden. Aufgrund der Toleranzen der Bauteile, vor allem der Kapazitäten, stellte sich ein Abgleich der beiden Stufen als sehr schwierig heraus, sodass sich die Charakteristik nicht wie theoretisch ermittelt ausprägte. Stattdessen verbreiterte sich der Durchlassbereich durch die Unterschiede der Mittenfrequenzen und die Dämpfung im Sperrbereich nahm ab.

### 3.3.4 Digitalfilter

Als letzte Methode wurde der Einsatz eines Digitalfilters zur Detektion des Synchronimpulses überprüft. Da die Auswertung mittels Analogfilter prinzipiell gute Ergebnisse lieferte, lag es nahe den größten Nachteil des analogen Aufbaus, die Bauteiltoleranz, durch Überführung des Prinzips in einen digitalen Filter, zu beseitigen. Dazu wurde exemplarisch ein idealer, diskreter Tiefpass in einen diskreten, kausalen Bandpass transformiert. Ausgangspunkt ist dabei die Übertragungsgleichung des Tiefpasses (Gleichung 3.13). Durch die inverse Fouriertransformation erhält man die Impulsantwort (Gleichung 3.14), welche die Basis für den Entwurf des Bandpasses ist.

$$H_{TP}(e^{\mathrm{j}\Omega}) = rect(\frac{2\Omega}{\pi}) \tag{3.13}$$

$$h_{TP}[k] = \frac{1}{2\pi} \int_{-\Omega_g}^{\Omega_g} 1 \cdot \mathrm{e}^{j\Omega k d\Omega} = \frac{\Omega_g}{\pi} si(k\Omega_g) \tag{3.14}$$

Durch eine Frequenztransformation gemäß Gleichung 3.15 erhält man durch Spiegelung und Verschiebung des Tiefpasses einen Bandpass mit doppelter Bandbreite. Der so entstandene Filter ist achsensymmetrisch zur Ordinate und somit akausal. Um einen kausalen Filter zu erhalten, muss die Impulsantwort um die halbe Filterlänge $(\frac{N}{2})$ verschoben werden (Gleichung 3.16).

$$h_{BP}[k] = h_{TP}[k] \cdot 2\cos(k\Omega_m) = \frac{\Omega_g}{\pi} si(k\Omega_g) \cdot 2\cos(k\Omega_m) \tag{3.15}$$

$$h_{BP}[k] = \frac{\Omega_g}{\pi} si((k-\frac{N}{2})\Omega_g) \cdot 2\cos((k-\frac{N}{2})\Omega_m) \tag{3.16}$$

Die Impulsantwort dieses kausalen Filter ist allerdings immer noch unendlich lang, da sie durch ein ideales System erzeugt wurde und ist somit nicht durch einen FIR reali-

| **Fensterfunktion** | **Breite des Hauptmaxmimums** (M ≙ Filterlänge) | **Dämpfung des Nebenmaximums** (relativ, in dB) |
|---|---|---|
| Rechteck | $\frac{4\pi}{M}+1$ | 13 |
| Barlett | $\frac{8\pi}{M}$ | 25 |
| Hanning | $\frac{8\pi}{M}$ | 31 |
| Hamming | $\frac{4\pi}{M}$ | 41 |
| Blackmann | $\frac{12\pi}{M}$ | 57 |
| Kaiser | variabel | variabel |

Tabelle 3.2: Eigenschaften ausgewählter Fensterfunktionen[10]

sierbar.

Um diese zu begrenzen, verwendet man so genannte Fensterfunktionen. Die einfachste stellt dabei das Rechteckfenster dar, welches den Filter durch Multiplikation mit der Impulsantwort, respektive Faltung mit dem Frequenzgang, auf eine gewünschte Länge $N+1$ begrenzt (Gleichung 3.17). Der Wert von $N$ entspricht dabei der Filterordnung und ist somit entscheidend für die Güte des Filters und die Gruppenlaufzeit des Signals, wobei der Filter $N+1$ *taps* umfasst.

Die Fensterfunktion verändert den Frequenzgang des Filters durch Begrenzung der Ordnung. Aus diesem Grund wurden im Laufe der Zeit verschiedene Funktionen definiert, welche unterschiedliche Eigenschaften besitzen. So können diese z.B. die Dämpfung im Sperrbereich und die Ausprägung der Nebenkeulen verändern. Beliebte Fensterfunktionen sind das Kaiser-Fenster, das Hamming-Fenster, sowie das Hann-Fenster, da diese eine besonders hohe Dämpfung der Nebenmaxima im Sperrbereich aufweisen. Eine Übersicht über die Eigenschaften verschiedener Fenster liefert Tabelle 3.2.

Im Anschluss an die „Fensterung“ des Filters, lassen sich die Koeffizienten berechnen (Gleichung 3.18). Aus den Filterkoeffizienten ergibt sich somit die Systemgleichung Gleichung 3.19.

$$h_{BP}[k] = \frac{\Omega_g}{\pi} si((k-\frac{N}{2})\Omega_g) \cdot 2\cos((k-\frac{N}{2})\Omega_m) \qquad k = 0,...,N \tag{3.17}$$

$$b_k = \frac{\Omega_g}{\pi} si((k-\frac{N}{2})\Omega_g) \cdot 2\cos((k-\frac{N}{2})\Omega_m) \qquad k = 0,...,N \tag{3.18}$$

$$H(z) = \frac{b_0 z^N + b_1 z^{N-1} + \ldots + b_N}{z^N} \tag{3.19}$$

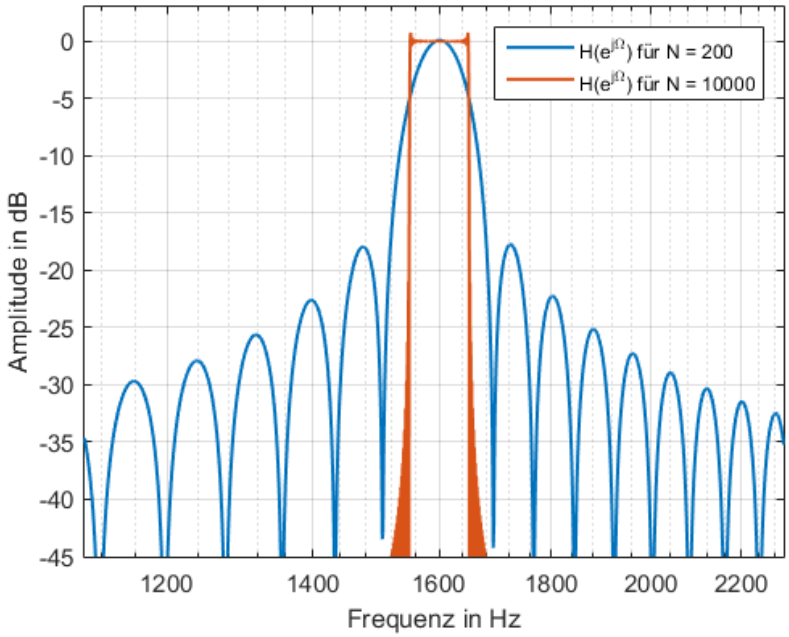



Abbildung 3.3: Frequenzgang zweier Bandpässe mit Rechteckfenster und verschiedener Filterordnung

Da der Speicherbedarf für FIR-Filter wesentlich höhere ist als bei IIR-Filtern, stellt die Filterlänge immer einen Kompromiss zwischen der Abweichung zum idealen Filter und dem nötigen Rechen- und Speicheraufwand dar. Abbildung 3.3 zeigt den gleichen Filter mit unterschiedlichen Längen. Es wird deutlich, dass eine höhere Filterordnung erheblich steilere Flanken erzeugt und näher am idealen Bandpass ist.

Neben der Flankensteilheit ist auch die Dämpfung im Sperrbereich wesentlich höher. Nachteilig wirkt sich allerdings das verstärkte Auftreten des *Gibbs'schen Phänomens* aus. Dies führt zu Überschwingungen bei steilflankigen Rechteckimpulsen und muss beim Entwurf digitaler Filter unbedingt beachtet werden. Die Wahl der Filterordnung ist demnach stark von der Anwendung und den gestellten Anforderungen abhängig. Die Ausprägung der Überschwingung, kann durch die Wahl der Fensterfunktion reduziert werden.

Aber auch Filter mit Ordnungen, die kaum Ähnlichkeit mit einem idealen Bandpass anmuten lassen, erfüllen häufig die gestellten Anforderungen. Wie im Diagramm zu sehen, besteht im Bereich des Hauptmaximas eine eher geringe Abweichung von der Idealen Form. Die geringere Breite der Durchlassdämpfung ist oft sogar erwünscht, da der Abfall des Pegels somit schneller von statten geht. Darüber hinaus ist die Dämpfung in

[10] [22]

den Nebenmaxima, im vorliegenden Beispiel über $15\mathrm{dB}$, für viele Anwendungen ausreichend.

Aufgrund der guten Anpassbarkeit des digitalen Filters an die technischen Randbedingungen und seiner hervorragenden Skalierbarkeit, wurde dieser für die Detektion des Synchronisationsimpulses gewählt. Die für die Anwendung abgestimmte Implementierung ist im nächsten Abschnitt zu finden (Unterabschnitt 3.4.2).

## 3.4 Implementierung

### 3.4.1 Antialiasing-Filter und Oversampling

Um den in Unterabschnitt 2.1.4 beschriebenen Effekt des Aliasings zu verringern, wurde dem Eingang des ADC ein Analogfilter mit der Grenzfrequenz $5KHz$ vorgeschaltet. Dieser soll Frequenzen oberhalb des Nutzbandes dämpfen. Nach den Vorbetrachtungen und praktischen Tests aus Kapitel 2 und Abschnitt 3.3 viel die Wahl auf einen aktiven Tiefpass zweiter Ordnung mit Sallen-Key-Topologie. Dieser bietet eine ausreichend hohe Flankensteilheit und erfordert eine vertretbare Komplexität bei der Implementierung. Es werden lediglich zwei Kondensatoren und zwei Widerstände, sowie ein Operationsverstärker benötigt. Darüber hinaus ist die Schaltung tolerant gegenüber Abweichungen bei Bauteilen, sie ist demnach ideal für den prototypischen Aufbau. Der Einsatz von Filtern höherer Ordnung zur Verbesserung der Qualität sollte in späteren Entwicklungsstadien in Betracht gezogen werden.

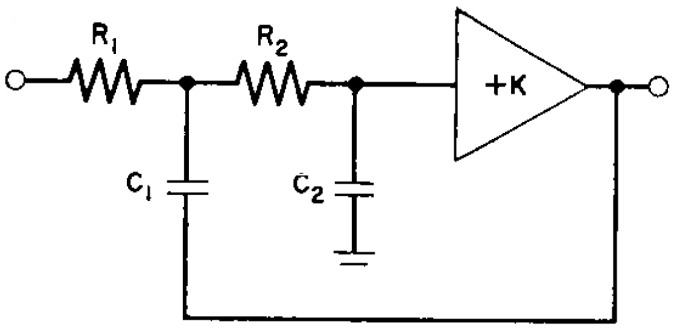



Abbildung 3.4: Aktive Filterstruktur nach Sallen und Key von 1955[11]

Durch Anwendung des Knotensatzes erhält man für die Schaltung in Abbildung 3.4 die Gleichung 3.20. Die Koeffizienten des komplexen Frequenzparameters entsprechen den Filterkoeffizienten und bestimmen somit die Charakteristik des Filters.

$$H(s) = \frac{1}{R_1 R_2 C_1 C_2 s^2 + C_2(R_1 + R_2)s + 1} \tag{3.20}$$

[11] [20, S. 3]

Die Laplace-Transformierte der Systemgleichung eines Tiefpasses zweiter Ordnung wird allgemein nach Gleichung 3.21 definiert und ist zugleich die in den Bildbereich transformierte Impulsantwort. Dabei entsprechen $\omega_c$ der Kreisfrequenz der Grenzfrequenz und die Koeffizienten $a_0$ und $a_1$ den Filterkoeffizienten.
Überführt man die Gleichung mit Hilfe der Korrespondenz aus Gleichung 3.22 in die Fourier-Transformierte, ergibt sich die frequenzabhängige Übertragungsgleichung des Filters (Gleichung 3.23).

$$H(s) = \frac{A_0 \omega_c^2}{a_0 s^2 + a_1 \omega_c s + \omega_c^2} \tag{3.21}$$

$$s = \mathrm{j}\omega \tag{3.22}$$

$$\begin{aligned} H(\mathrm{j}\omega) &= \frac{A_0 \omega_c^2}{a_0 (\mathrm{j}\omega)^2 + a_1 \omega_c \mathrm{j}\omega + \omega_c^2} \\ &= \frac{A_0 \omega_c^2}{-a_0 \omega^2 + a_1 \omega_c \mathrm{j}\omega + \omega_c^2} \end{aligned} \tag{3.23}$$

Für ein lineares Verhalten im Übertragungsbereich, welches bei der Verarbeitung von Audiosignalen von Vorteil ist, empfiehlt sich die Butterworth-Charakteristik. Dies besitzt eine äußerst geringe Welligkeit im Durchlassbereich, wodurch die Verzerrung des Signals minimiert wird. Hierfür sind die Filterkoeffizienten wie folgt zu wählen:

$$\begin{aligned} a_0 &= 1 \\ a_1 &= \sqrt{2} \end{aligned}$$

Die vollständige Gleichung für den Frequenzgang des Tiefpasses zweiter Ordnung nach Sallen und Key lässt sich demnach mit Gleichung 3.24 beschreiben und wie in Abbildung 3.5 grafisch darstellen.
Zu sehen ist ein Tiefpass mit der Grenzfrequenz von $f_c = 5\mathrm{kHz}$ und der Übertragungsverstärkung von $A_0 = 1$. Neben der Grenzfrequenz von $5\mathrm{kHz}$, bei der die Verstärkung $-3\mathrm{dB}$ beträgt, ist auch der Abfall der selbigen von $40\mathrm{dB}$ pro Dekade zu erkennen.

$$H(\mathrm{j}\omega) = \frac{\omega_c^2}{-\omega^2 + \sqrt{2}\omega_c \mathrm{j}\omega + \omega_c^2} \tag{3.24}$$

Zusätzlich werden die spektralen Wiederholungen durch Oversampling mit dem Faktor zwei in höhere Frequenzbereiche verschoben, sodass die Dämpfung des Filters trotz

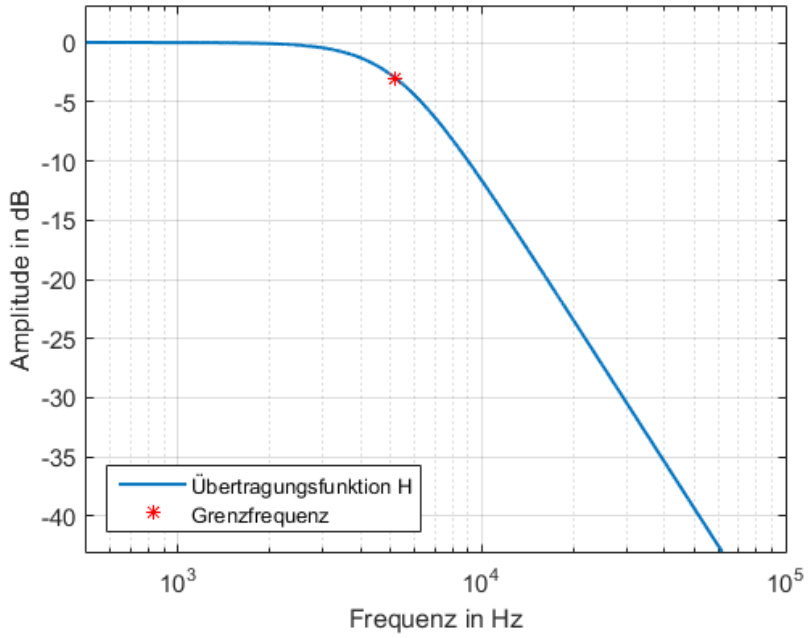



Abbildung 3.5: Frequenzgang eines aktiven Tiefpasses zweiter Ordnung nach Sallen und Key

der geringen Ordnung ausreichend hoch ist. Der Antialiasing-Filter soll dabei Störungen im Frequenzspektrum dämpfen, welche nicht im Basisband liegen, später allerdings durch Aliasing in dieses gemischt werden könnten.
Aus Abbildung 3.6 wird deutlich, dass die erste spektrale Wiederholung des Nutzsignals bei $12,6kHz$ beginnt. Setzt man diesen Wert in Gleichung 3.24 ein, ergibt sich eine Dämpfung von $|H(\mathrm{j}\omega)|_{dB} = 16.1624\mathrm{dB}$. Die Amplitude ist bei dieser Frequenz somit bereits auf weniger als 20% des Basisbandes abgesunken. Für eine Weiterentwicklung wäre ein Filter höherer Ordnung zur Verbesserung des SNR zu empfehlen, für diesen prototypischen Aufbau, soll dieser allerdings genügen.

Führt man zwischen Gleichung 3.20 und Gleichung 3.21 einen Koeffizientenvergleich durch, erhält man die für die Dimensionierung der Bauteile benötigten Gleichungen (Gleichung 3.25), wobei $C_1$ frei wählbar ist und gemeinsam mit $C_2$ lediglich die Bedingung aus Gleichung 3.26 erfüllen muss.

$$R_{1,2} = \frac{a_1 C_2 \pm \sqrt{a_1^2 C_2^2 - 4 a_0 C_1 C_2}}{2\omega_c C_1 C_2} \tag{3.25}$$

$$C_2 \geq C_1 \frac{4a_0}{a_1^2} \tag{3.26}$$

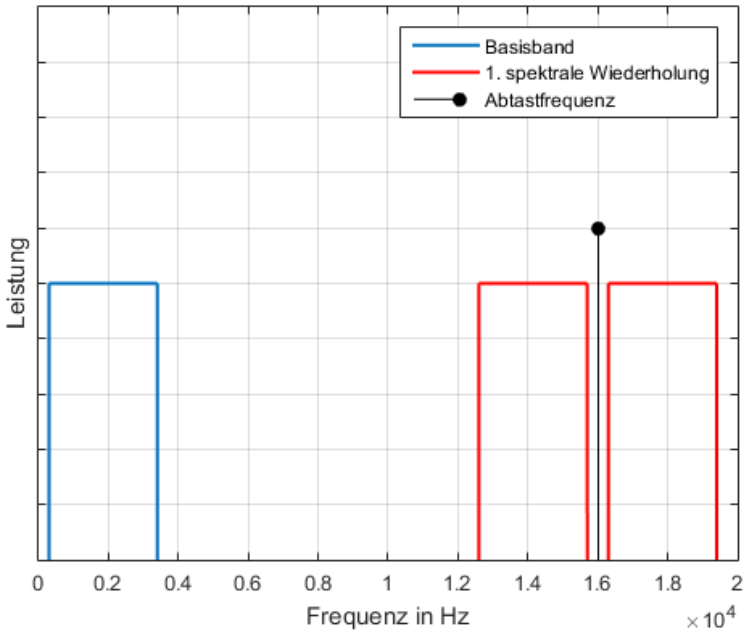



Abbildung 3.6: Ausschnitt des Spektrums des mit 16 kHz abgetasteten Signals

Für $C_1 = 4,7\text{nF}$ ergeben sich somit folgende, an die E-Reihe angepasste, Werte:

$$\begin{aligned} C_1 &= 4,7\,\text{nF} \\ C_2 &= 22\,\text{nF} \\ R_1 &= 1164 \approx 2200\,\Omega \\ R_2 &= 8413 \approx 8200\,\Omega \end{aligned}$$

Setzt man diese Werte in eine der Gleichungen für die Koeffizienten ein, ergibt sich eine reale Grenzfrequenz von $4.6\text{kHz}$ (Gleichung 3.27). Diese liegt zwar um einiges tiefer als die ursprünglich gewählte Frequenz, ist allerdings immer noch groß genug, um das Signal am oberen Ende des Basisbandes nicht zu stark zu dämpfen.

$$\begin{aligned} a_1 &= \omega_c C_1 (R_1 + R_2) \\ f_c &= \frac{a_1}{2\pi C_1 (R_1 + R_2)} \\ &= \frac{1.4142}{2\pi \cdot 4.7\text{nF} \cdot (2200\Omega + 8200\Omega)} \\ &= \underline{\underline{4605\text{kHz}}} \end{aligned} \tag{3.27}$$

### 3.4.2 Digitalfilter zur Impulsdetektion

Anhand der Untersuchungen aus Abschnitt 3.3 konnte festgestellt werden, dass ein digitaler FIR-Filter am besten den Anforderungen und Voraussetzungen der zu entwi-

ckelnden Anwendung genügt. Für den verwendeten Mikrocontroller mit ARM Cortex-M3 Architektur steht eine DSP-Bibliothek von ARM zur Verfügung, welche auch die Implementierung von FIR-Filtern unterstützt.

Die Koeffizienten wurden gemäß den Herleitungen und Beispielen in Unterabschnitt 3.3.4 berechnet. Die optimale Filterlänge, sowie die zu wählende Fensterfunktion wurden zur Abschätzung der Implementierbarkeit vorab mit dem Matlab-Tool *fdatool* ermittelt. Anschließend wurden die Ergebnisse theoretisch verifiziert. Demnach wird für ein 120Hz breites Hauptmaximum, welches einer Bandbreite von ca. 50 Hz entspricht, eine Filterlänge von mindestens 315 Taps benötigt, wobei die Sperrdämpfung 24.8075 dB beträgt. Als Fensterfunktion wurde aufgrund der hohen Variabilität das Kaiser-Fenster gewählt.

Das Kaiser-Fenster wurde 1980 von *James F. Kaiser* und *Ronald W. Schafer* im Journal *IEEE TRANSACTIONS ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING* [11] beschrieben und ermöglicht die Anpassung der Breite des Hauptmaximums und der Dämpfung des Nebenmaximums in Abhängigkeit von Bandbreite, Filterordnung und einem Koeffizienten $\beta$. Es ist durch Gleichung 3.28 definiert.

$$w_{Kaiser}[k] = \frac{I_0(\beta\sqrt{1-(\frac{2(k-\frac{N}{2})}{N})^2})}{I_0(\beta)} \qquad k = 0,...,N \tag{3.28}$$

Je größer der Faktor $\beta$ ist, desto höher ist die Dämpfung der Nebenmaxima und desto größer ist die Breite der Hauptmaxima. Für die vorliegende Anwendung wird eine geringe Breite des Hauptmaximums benötigt, um die Frequenz so genau wie möglich identifizieren zu können. Die Dämpfung der Nebenmaxima ist von weniger großer Bedeutung, da die Differenzierung dieser algorithmisch erfolgen kann. Das *fdatool* lieferte für den Parameter den Wert $\beta = 1.2973$. Aufgrund seiner Ordnung besitzt der Filter eine Länge von 315 Taps. Die Koeffizienten und die daraus resultierende Übertragungsgleichung können Anlage B entnommen werden. Die graphische Darstellung des Übertragungsverhaltens ist in Abbildung 3.7 zu sehen. Klar zu erkennen ist der sehr schmale Übertragungsbereich von wenigen Herz, sowie der Übergangsbereich, welcher bereits bei weniger als 60 Hz abseits des Maximums einen Wert von -20dB unterschreitet.

Der Nachweis der Filterparameter erfolgt über die in Unterabschnitt 3.3.4 hergeleiteten, sowie die von *Kaiser* und *Schafer* aufgestellten Gleichungen. Mit der durch *Matlab* ermittelten Sperrdämpfung von $a_s = 24.8075\text{dB}$ ergibt sich nach Gleichung 3.29[12] eine Filterlänge von $M = 315$ und somit eine Ordnung von $N = 314$. Der nun noch benötigte Kaiser-Faktor $\beta$ berechnet sich durch Gleichung 3.30 zu $\beta = 1.2973$, womit alle Parameter des Kaiser-Fensters, sowie die Ordnung des Filters, bestimmt sind.

[12] [22]

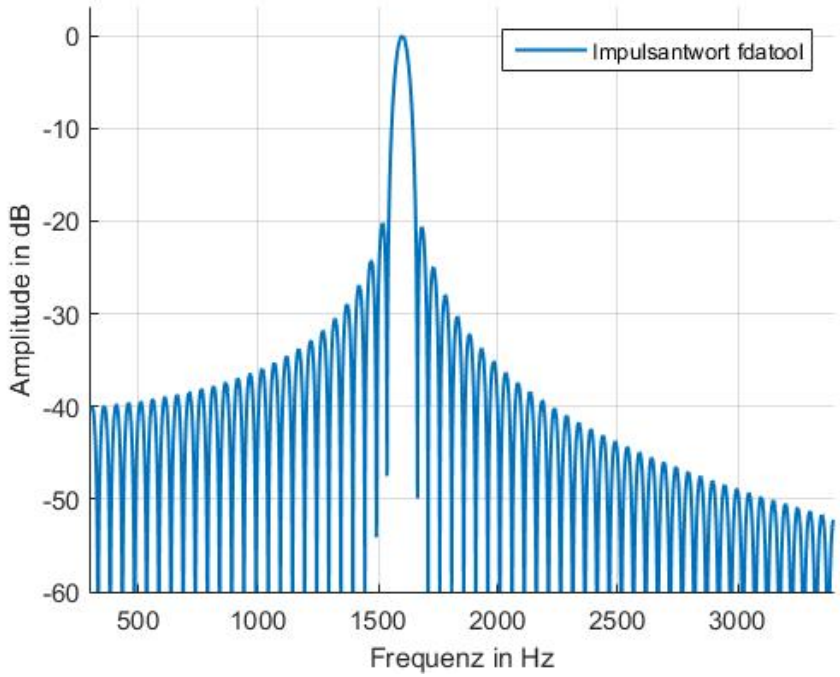



Abbildung 3.7: Impulsantwort des mit Hilfe von Matlab berechneten FIR-Filters

$$M = \begin{cases} \frac{f_s(a_s - 7.95)}{14.36 T_r} + 1 & a_s > 21 \\ \frac{0.922 f_s}{T_r} + 1 & a_s \leq 21 \end{cases} \tag{3.29}$$

$$= \frac{16000\text{Hz}(24.8075\text{db} - 7.95)}{14.36 \cdot 60\text{Hz}} + 1$$
$$= \underline{\underline{314.0455}}$$

$$\beta = \begin{cases} 0.1102(a_s - 8.7) & a_s > 50 \\ 0.5842(a_s - 21)^{0.4} + 0.0788(a_s - 21) & 21 \leq a_s \leq 50 \\ 0 & a_s < 21 \end{cases} \tag{3.30}$$

$$= 0.5842(24.8075 - 21)^{0.4} + 0.0788(24.8075 - 21)$$
$$= \underline{\underline{1.2973}}$$

Durch Multiplikation dieser Fensterfunktion mit der Filterfunktion Gleichung 3.17 ergibt sich die vollständige Gleichung $h_{BP,Kaiser}[k]$ des Digitalfilters (Gleichung 3.31).

$$h_{BP,Kaiser}[k] = \frac{\Omega_g}{\pi} si((k-\frac{N}{2})\Omega_g) \cdot 2\cos((k-\frac{N}{2})\Omega_m) \cdot \frac{I_0(\beta\sqrt{1-(\frac{2(k-\frac{N}{2})}{N})^2})}{I_0(\beta)} \quad k = 0,...,N \tag{3.31}$$

Abschließend, muss der Filter noch skaliert werden, um eine Durchgangsdämpfung von 0 zu erreichen. Dazu werden die spektralen Anteile der Filterkoeffizienten an der Mittenfrequenz aufsummiert und anschließend mit den Koeffizienten selbst skaliert. Der nun resultierende Filter weist bei der Mittenfrequenz Einheitsverstärkung und somit weder eine Verstärkung, noch eine Dämpfung auf (Gleichung 3.32).

$$g = |\sum_{k=0}^{N} b_k e^{j\Omega_m k}|$$

$$h_{BP,Kaiser}[k] = \frac{\frac{\Omega_g}{\pi} si((k-\frac{N}{2})\Omega_g) \cdot 2\cos((k-\frac{N}{2})\Omega_m) \cdot \frac{I_0(\beta\sqrt{1-(\frac{2(k-\frac{N}{2})}{N})^2})}{I_0(\beta)}}{g} \quad k = 0,...,N \tag{3.32}$$

Abbildung 3.8 zeigt sowohl die Impulsantwort des mit Matlab bestimmten Filters, als auch die, des berechneten Filters - es ist kein Unterschied sichtbar.

### Das Q-Format als Festkommaformat und seine Konvertierung

Viele Mikrocontroller, so auch der Cortex-M3, verfügen über keinen Gleitkommarechner und können Gleitkommaberechnungen somit nur durch Berechnungen mittels Festkommazahlen annähern. Die Konvertierung der Gleitkommawerte in Festkommazahlen erfordert allerdings eine große Anzahl an Operationen und somit viel Rechenzeit. So benötigt der Cortex-M3 für die Grundrechenarten mit Gleitkommazahlen zwischen 53 und 76 Zyklen, wohingegen Festkommazahlen bereits nach 1 bis 5 Zyklen berechnet werden können[13]. Der Befehlssatz des Cortex-M3 besitzt speziell dafür optimierte, atomare Befehle und Register, welche diese äußerst schnelle Berechnugn erlauben. Außerdem belegen Gleitkommazahlen beim verwendeten Controller 4 Byte Speicher, wohingegen der Speicherverbrauch des Q-Formates in Abhängigkeit der Auflösung frei wählbar ist.

[13] Anzahl der Zyklen (Differenz des Cyclecounters) inklusive der Lese- und Speicheroperationen der Variablen

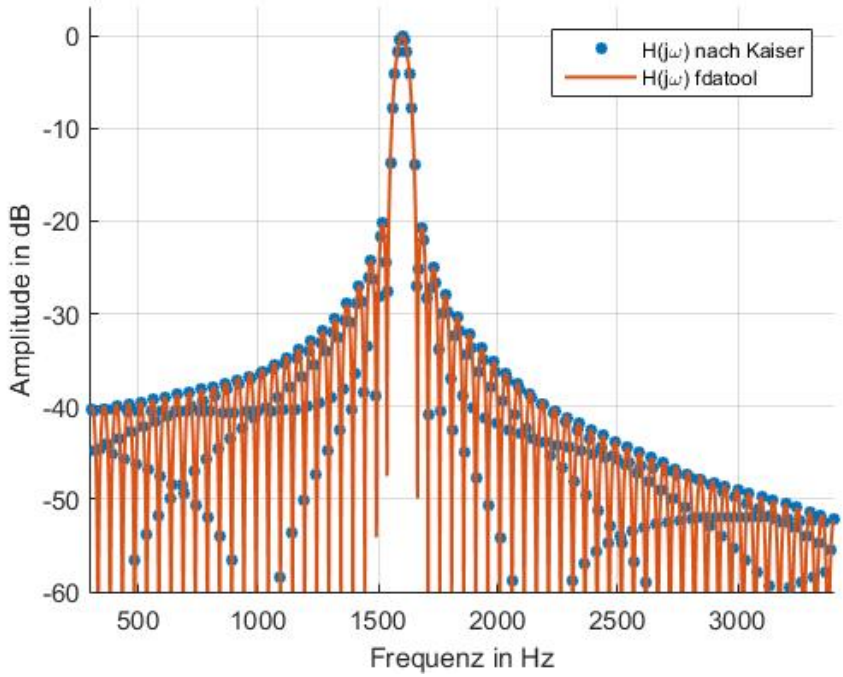



Abbildung 3.8: Frequenzgänge des berechneten und mit Matlab modellierten Filters

Da der Cortex-M3 prinzipiell nicht als digitaler Signalprozessor (DSP) ausgelegt ist und somit nur über einen eingeschränkten Befehlssatz dafür verfügt, wurde versucht die Rechenzeit so gut wie möglich zu reduzieren. Aus diesem Grund wurden die Rechenoperationen im sogenannten Q15-Format umgesetzt.

| **Datentyp** | **Multiplikation** | **Division** | **Addition** | **Subtraktion** |
|---|---|---|---|---|
| **Integer** | 1 | 5 | 1 | 1 |
| **Float** | 53 | 76 | 54 | 61 |

Tabelle 3.3: Anzahl der für Grundrechenarten benötigten Zyklen des LPC1768

Das Q-Format wird in vielen Anwendungen als Zahlenformat für die Festkommaarithmetik verwendet und daher auch von der CMSIS-DSP Bibliothek unterstützt. Die Schreibweise Qm.n sagt aus, dass die konvertierte Zahl m Bit für die Vorkommastellen und n Bit für die Nachkommastellen verwendet. Für $m = 1$ kann die Bezeichnung auf Qn verkürzt werden. Q15 verwendet somit ein Bit für die Vorkomma- und 15 Bit für die Nachkommastellen. Es ermöglicht demnach Werte zwischen -1 und 1 bei einer Auflösung von $2^{-15}$.

Die Konvertierung von Gleitkommazahlen in das Q-Format und umgekehrt erfolgt gemäß den Gleichungen 3.33. Dabei werden die Q-Zahlen als Integer dargestellt, was die Nutzung der dafür vorhandenen Befehle ermöglicht. Zu beachten ist, dass die Bibliothek CMSIS-DSP nur Q-Formate mit einem Bit für die Vorkommastelle zulässt und

Gleitkommazahlen somit auf den Wertebereich $[-1;1]$ normiert werden müssen.

$$\begin{aligned} Q &\approx float \cdot 2^n \\ float &= Q \cdot 2^{-n} \end{aligned} \tag{3.33}$$

Die konvertierten und quantisierten Filterkoeffizienten können Anlage B entnommen werden.

### 3.4.3 Rekonstruktionsfilter

Wie bereits in Unterabschnitt 2.1.5 beschrieben, wird am Analogausgang des Controllers ein Filter zur Rekonstruktion des gewünschten Informationssignals benötigt. Dieser wurde identisch zum Anti-Aliasing-Filter am Analogeingang des Controllers entworfen. Auch hier könnte die Qualität des Ausgangssignals noch verbessert werden, wenn die Ordnung des Filters erhöht wird, allerdings rechtfertigt die Steigerung den Aufwand in diesem Entwicklungsstadium noch nicht. Abbildung 3.9 zeigt einen Vergleich zwischen einem mit $16\mathrm{kHz}$ abgetastetem Sinus mit einer Frequenz von $3,6\mathrm{kHz}$, sowie zwei rekonstruierten Signalen. Die Signalfrequenz spiegelt dabei die höchste, im Spektrum der Information vorkommende, Frequenz wieder. Das rote Signal wurde mit einem Filter zweiter Ordnung bei einer Grenzfrequenz von $5\mathrm{kHz}$ gefiltert, das blaue Signal mit einem Filter vierter Ordnung. Es wird deutlich, dass zwar noch ein minimaler, jedoch kein eklatanter Unterschied zwischen den Kurven mehr festzustellen ist. Der Entschluss fiel daher auf den Filter zweiter Ordnung. Für die Schaltung nach Abbildung 3.4 ergeben sich demnach folgende Werte für die Bauelemente:

$$\begin{aligned} C_1 &= 4,7\,\mathrm{nF} \\ C_2 &= 22\,\mathrm{nF} \\ R_1 &= 2200\,\Omega \\ R_2 &= 8200\,\Omega \end{aligned}$$

.

Führt man den Vergleich bei der unteren Grenzfrequenz des Basisbandes von $300\mathrm{Hz}$ durch, ergeben sich keinerlei sichtbare Unterschiede mehr Abbildung 3.10. Der Filter scheint somit einen guten Kompromiss zwischen Komplexität und Qualität zu erzielen.

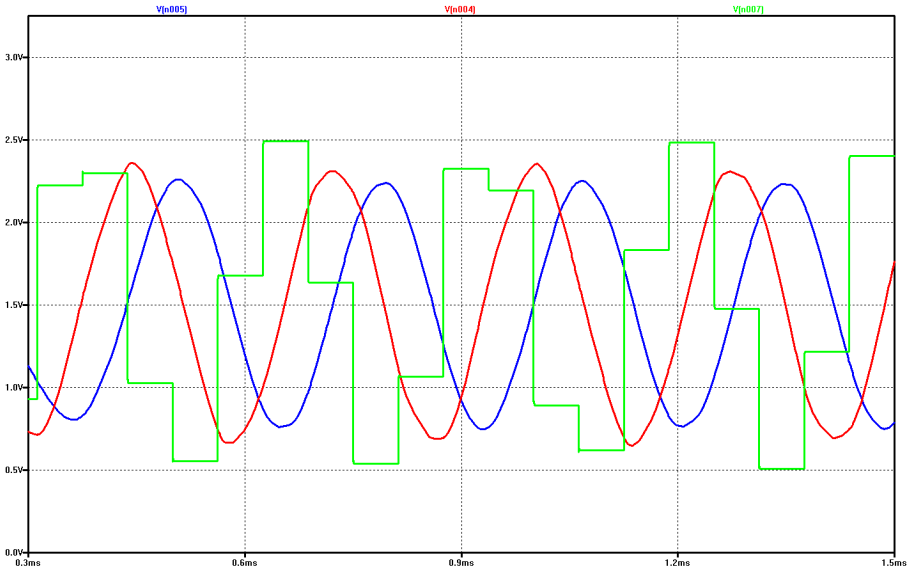



Abbildung 3.9: Gegenüberstellung der Ausgangssignale von Rekonstruktionsfiltern zweiter und vierter Ordnung bei 3,4kHz

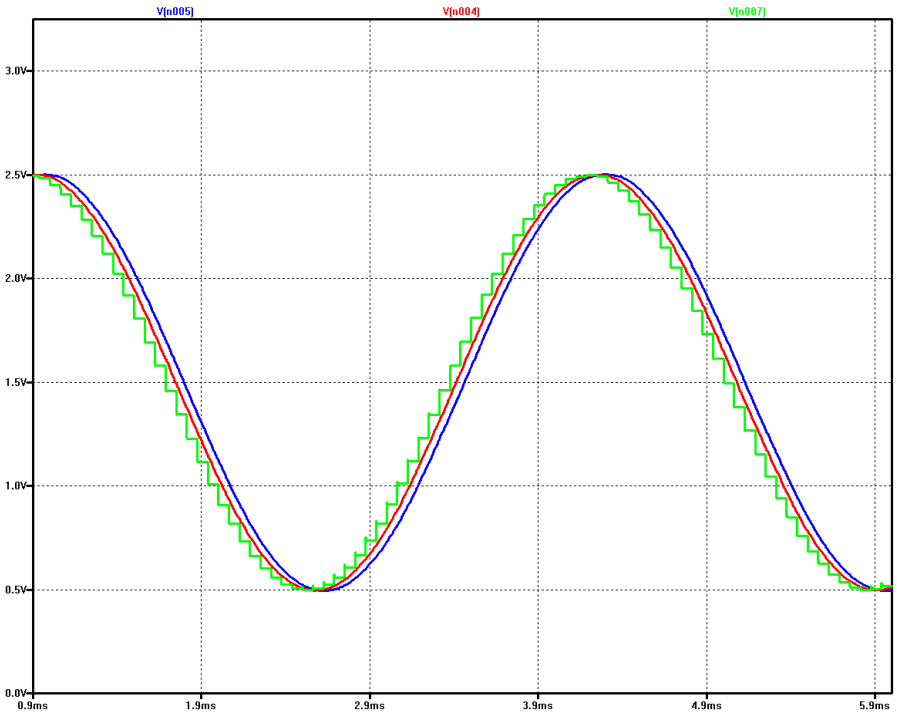



Abbildung 3.10: Gegenüberstellung der Ausgangssignale von Rekonstruktionsfiltern zweiter und vierter Ordnung bei 300Hz

# 4 Versuchsaufbau für analoge Sprachübertragung

## 4.1 Allgemeines

Wie einleitend bereits erwähnt, wird für die Realisierung der Anwendung der LPC1768 von NXP, ein Mikrocontroller mit Cortex-M3 Architektur, auf einem Entwicklungsboard des Herstellers *Elektronikladen* verwendet (Abbildung 4.1). In diesem Kapitel soll gezeigt werden, wie die gestellten Anforderungen und theoretisch hergeleiteten Randbedingungen und Parameter auf den vorgegebenen Controller übertragen und in die Praxis umgesetzt wurden. Dazu wurden unter anderem wiederum *Matlab* und die Entwicklungsumgebung *µVision 5* von *KEIL* verwendet.

## 4.2 Hardware

Der schaltungstechnische Aufbau wurde in sechs wesentliche Baugruppen unterteilt, welche sich wie folgt gliedern:

- Eingangsstufe
- Anti-Aliasing-Filter
- D/A-Wandler und DSP
- Rekonstruktionsfilter
- Ausgangsstufe
- Carrierboard

Die Ein- und Ausgangsstufe, sowie der D/A-Wandler befinden sich auf einer gesonderten Lochrrasterplatine, welche über Buchsenleisten mit dem Entwicklungsboard, dem Carrierboard, verbunden ist. Auf dem Board selber befindet sich eine weitere Platine mit dem LPC1768 (Abbildung 4.2), sowie einiges an Peripherie, u.a. Taster, LEDs und Datenschnittstellen, welche die Interaktion mit dem Controller gestatten. Die Spannungsversorgung des Aufbaus erfolgt über ein 12V Schaltnetzteil, die einzelnen Baugruppen werden über bereits verbaute Spannungsregler versorgt. Der vollständige Schaltplan der entwickelten Schaltung ist in Anlage A zu finden. Die Schaltpläne der proprietären Hardware befindet sich in den dazugehörigen Handbüchern in Anlage C.

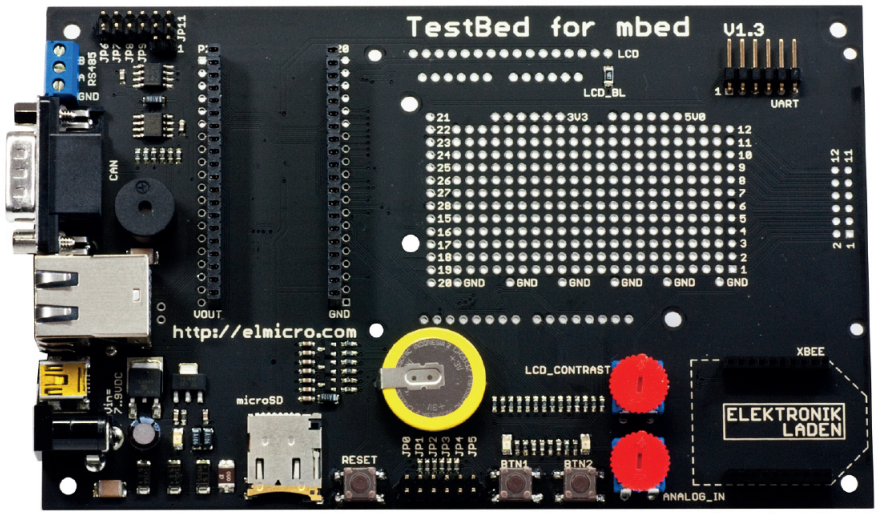



Abbildung 4.1: Entwicklungsboard *testBed* von *Elektronikladen*

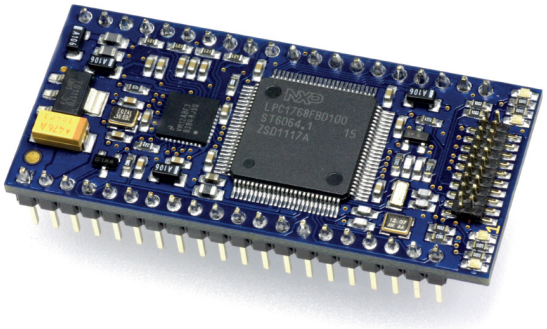


Abbildung 4.2: Microcontrollerunit Chip1768 mit LPC1768

### 4.2.1 Eingangsstufe

Die Eingangsstufe dient der Einspeisung der Audiosignale, sowie deren Anpassung an das Mikrocontrollersystem, welches einen Eingangsspannungsbereich von $0\text{V}$ bis $3.3\text{V}$ zulässt. Dazu verfügt sie über einen Cinch-Steckverbinder, über den Audiogeräte an den Aufbau gekoppelt werden können. Anschließend findet eine gleichspannungsmäßige Entkopplung mit Hilfe eines Kondensators statt, um über den folgenden Spannungsteiler eine Gleichspannung von 1.42V einprägen zu können. Diese dient als virtuelle Masse für die verbauten Operationsverstärker, welche eine asymmetrische Spannungsversorgung besitzen. Der Eingangskondensator und die wechselspannungsmäßige Parallelschaltung der Spannungsteilerwiderstände bilden einen Hochpass. Sie wurden daher so dimensioniert, dass die untere Grenzfrequenz des Übertragungskanals von 300Hz nicht überschritten wird. Dadurch werden Störungen unterhalb dieser Frequenz bereits hier gefiltert, ohne dass das Basisband davon beeinträchtigt wird. Zuletzt erfolgt eine Impedanzwandlung durch einen Spannungsfolger, um die Eingangsschaltung vom übri-

gen Netzwerk zu entkoppeln und eine belastbare Signalquelle für die weitere Schaltung zur Verfügung zu stellen.

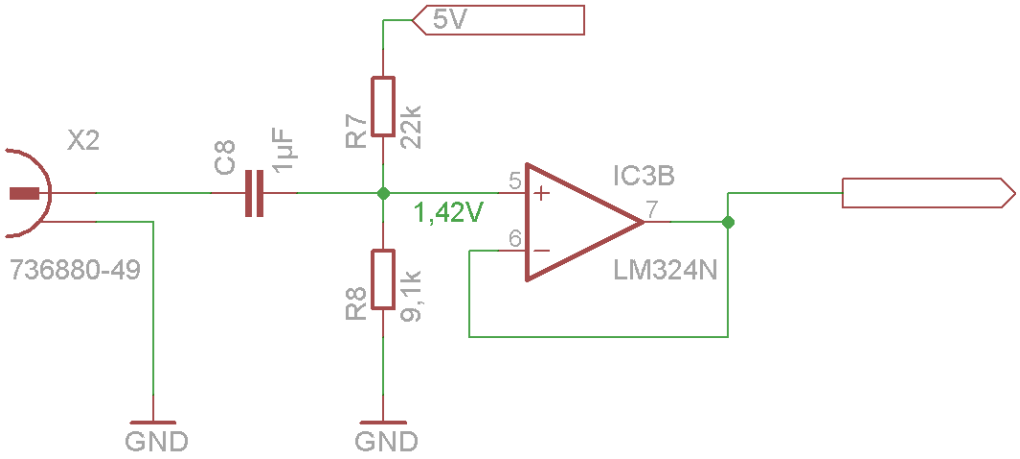



Abbildung 4.3: Eingangsstufe mit Koppelkondensator, Spannungsteiler und Impedanzwandler

### 4.2.2 Anti-Aliasing-Filter

Der Anti-Aliasing-Filter wurde gemäß Unterabschnitt 3.4.1 gestaltet und dient der Reduktion von Störungen außerhalb des Basisbandes. Um eine zu hohe Dämpfung des Informationssignales zu vermeiden, wurde die Grenzfrequenz auf $4,6\mathrm{kHz}$ festgelegt. Gemeinsam mit dem Oversamplingfaktor von 2, und der Dämpfung des Filters von $40\mathrm{dB}$ pro Dekade lässt sich übermäßiges Aliasing so ausreichend gut unterdrücken. Auch dieser Schaltungsteil wird abschließend mit einem Impedanzwandler von nachfolgenden Bauelementen entkoppelt, um Wechselwirkungen zwischen verschiedenen Baugruppen zu vermeiden.

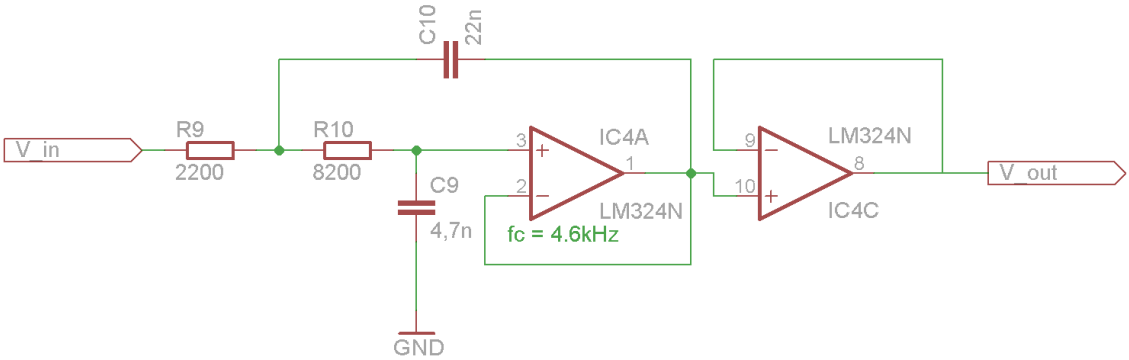



Abbildung 4.4: Anti-Aliasing-Filter mit Impedanzwandler zur Entkopplung

### 4.2.3 A/D-Wandler, DSP und D/A-Wandler

In dieser Baugruppe wurde ein externer A/D-Wandler der Firma *Microchip* verwendet. Vorangegangene Entwicklungen und Versuche der Fachgruppe für Kommunikationstechnik der Hochschule Mittweida haben gezeigt, dass der On-Chip-Wandler des LPC1768 sehr störanfällig ist und stochastisch *glitches* erzeugt. Um dieser bekannten Fehlerquelle vorzubeugen, wurde stattdessen auf den MCP3001 zurückgegriffen. Es handelt sich dabei um einen 10Bit-Wandler, welcher über *SPI* mit dem Mikrocontroller kommuniziert. Dieser erreicht eine maximale Datenrate von 250ksp/s und ist somit ausreichend für die anvisierte Abtastfrequenz von $16\mathrm{kHz}$. Zur Entstörung des IC, befindet sich am Pin der Spannungsversorgung ein entsprechender Kondensator. Der digitale Eingangspin des SPI-Interfaces (MOSI) bleibt unbelegt, da der Wandler keine Daten empfängt, sondern nur sendet.

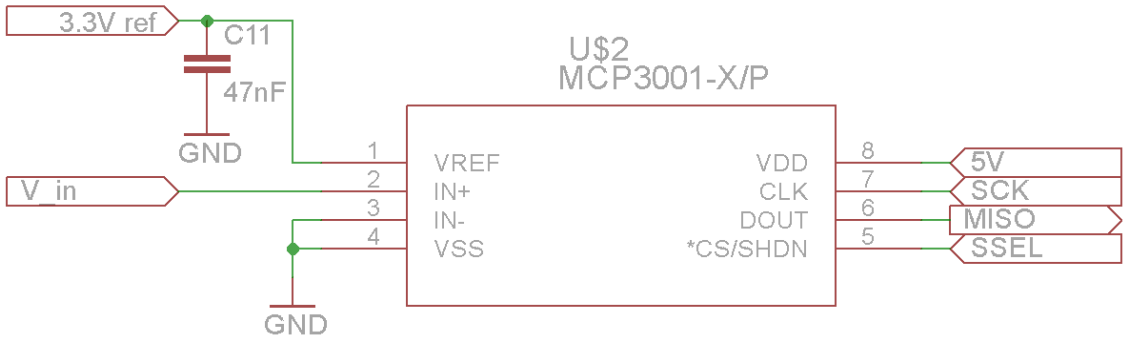



Abbildung 4.5: A/D-Wandler MCP3001

Der LPC1768 führt anschließend die digitale Signalverarbeitung durch. In diesem Fall wird der in Unterabschnitt 3.4.2 beschriebene Digitalfilter auf das digitalisierte Signal angewendet um den Synchronimpuls zu detektieren. Das Audiosignal selbst wird digitalisiert, verschlüsselt (Vgl. Abschnitt 4.3.3) und anschließend über den D/A-Wandler des LPC1768 ausgegeben.
Der Digital-Analog-Wandler des LPC1768 bietet mit einer Auflösung von $10\mathrm{Bit}$ bei einer Referenzspannung von $3.3\mathrm{V}$, und einer maximalen Taktfrequenz von $1\mathrm{MHz}$ eine ausreichende Leistungsfähigkeit.

### 4.2.4 Rekonstruktionsfilter

In Unterabschnitt 2.1.5 und Unterabschnitt 3.4.3 wurden sowohl die Gründe, als auch die Anforderungen an einen solchen Filter beschrieben. Er gleicht bei dieser Anwendung im Aufbau dem Anti-Aliasing-Filter aus Unterabschnitt 4.2.2, befindet sich allerdings am Analogausgang des Mikrocontrollers, dem D/A-Wandler und beseitigt die

durch die Abtastung entstandenen Wiederholungen des Basisbandes im Spektrum.

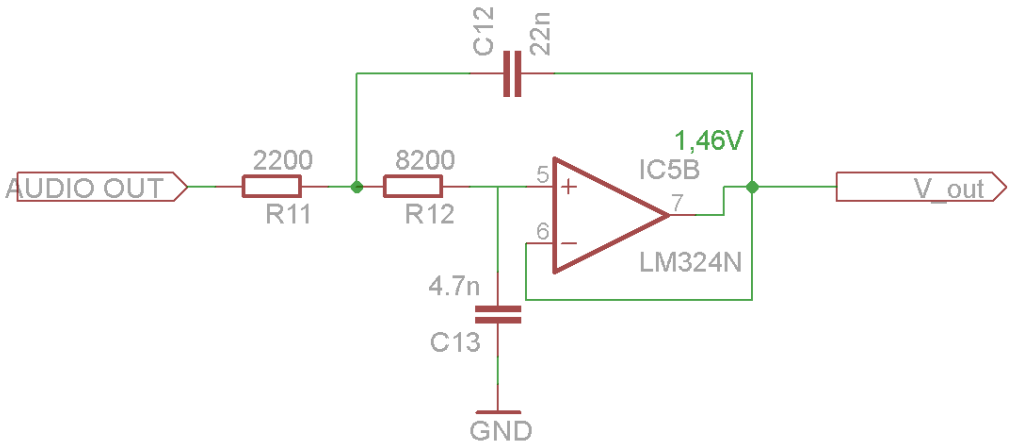



Abbildung 4.6: Rekonstruktionsfilter am Analogausgang des Mikrocontrollers

### 4.2.5 Ausgangsstufe

Die Ausgangsstufe stellt die letzte Baugruppe im Signalzweig der Schaltung dar. Sie ermöglicht die Kopplung externer (Wiedergabe-) Geräte an die Schaltung. Dazu kommt auch hier ein Operationsverstärker als Impedanzwandler zum Einsatz, der die Schaltung entkoppelt und den benötigten Strom am Ausgang für niederohmige Verbraucher liefert. Zusätzlich erfolgt eine Gleichspannungsentkopplung mit einem Kondensator. Mit einem maximalem Ausgangsstrom von $40\mathrm{mA}$, kann die Schaltung Verbraucher mit einer minimalen Eingangsimpedanz von bis zu 100 Ω treiben. Üblicherweise besitzen Audioeingänge eine Eingangsimpedanzen im Bereich von einigen Kiloohm.

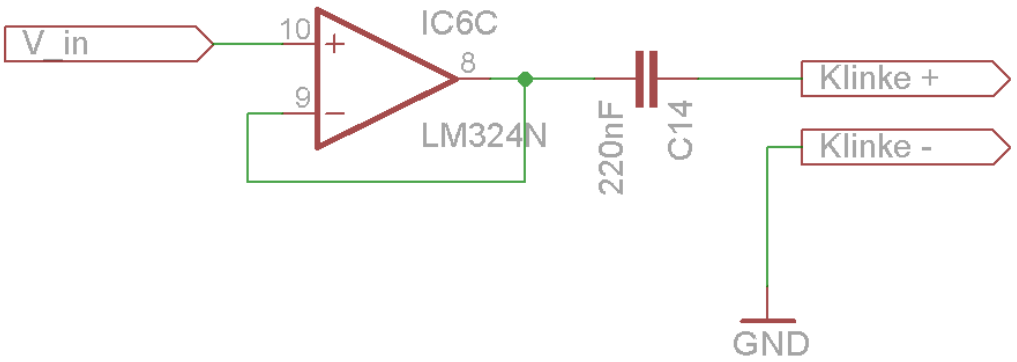



Abbildung 4.7: Ausgangsstufe bestehend aus Impedanzwandler und Koppelkondensator

### 4.2.6 Ethernetschnitstelle

Die Kommunikation über Ethernet wird über einen internen *Ethernet Media Access Controller* (*EMAC*) und einem externen Controller (*DP83848J*), welcher als PHY fungiert,

ermöglicht. Der EMAC steuert den Medienzugriff gemäß dem OSI-Modell auf Schicht 2 des OSI-Referenzmodells und dem Standard IEEE 802.3, welcher die Grundlage für Ethernet bildet. Der über das *Reduced Media Independent Interface* (*RMII*) angebundene PHY-Controller hingegen codiert und decodiert den Datenstrom entsprechend den Vorgaben des Standards für Schicht 1. Im Anschluss an den Ethernet-Transceiver folgt die Verbindung zum physischen Übertragungsmedium, welche als *Medium Dependent Interface* (*MDI*) bezeichnet wird. Sofern ausreichend Rechenzeit für die Verarbeitung der zu sendenden und zu empfangenden Daten bereit steht, können Daten mit bis zu $100\mathrm{MBit/s}$ im Duplex-Betrieb übertragen werden. Wird die Schnittstelle mit einem anderen Endgerät, z.B. einem PC verbunden, muss das Gerät entweder $AutoMDI-X$ beherrschen oder ein Crossoverkabel verwendet werden. Abbildung 4.8 zeigt eine schematische Darstellung der kompletten Baugruppe und der Verschaltung. Ein detaillierter Schaltplan des Evaluationsboards kann dem Handbuch in Anhang C entnommen werden.

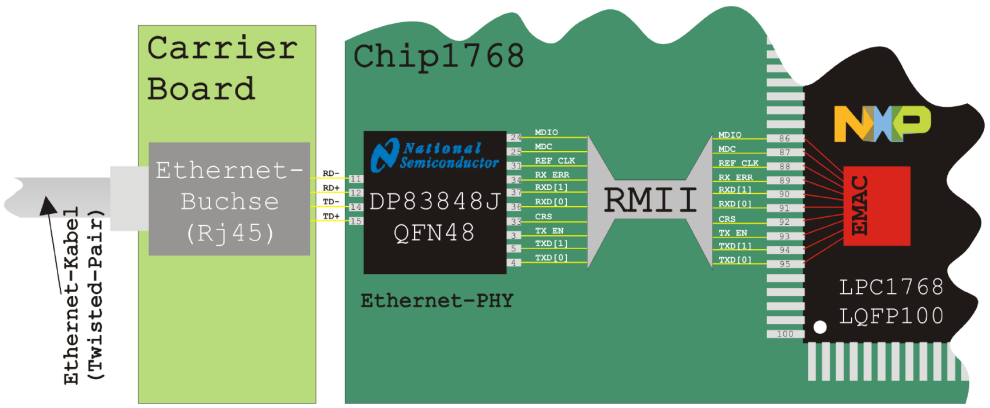



Abbildung 4.8: Übersicht der Ethernetschnittstelle mit EMAC, PHY und Buchse[14]

Wie aus Abbildung 4.8 hervorgeht, muss die Ethernetbuchse auf einem gesondertem Carrier-Board bereitgestellt werden. Für die Applikation geschieht dies über das verwendete Entwicklungsboard *testbed*, welches bereits eine entsprechende Buchse vom Typ RJ45 verbaut hat. Der Anschluss erfolgt über die, zum *Chip1768* kompatiblen, Buchsen- und Stiftleisten. Wie im Schaltplan in Abbildung 4.9 zu sehen, besitzt der Ethernetanschluss neben der Buchse selbst noch zwei LEDs, welche den Status der Verbindung anzeigen. *LINKACT* signalisiert eine verbundene Gegenstelle, *LINKSPD* wird aktiv, sobald Daten übertragen werden.

[14] [5, S. 14]

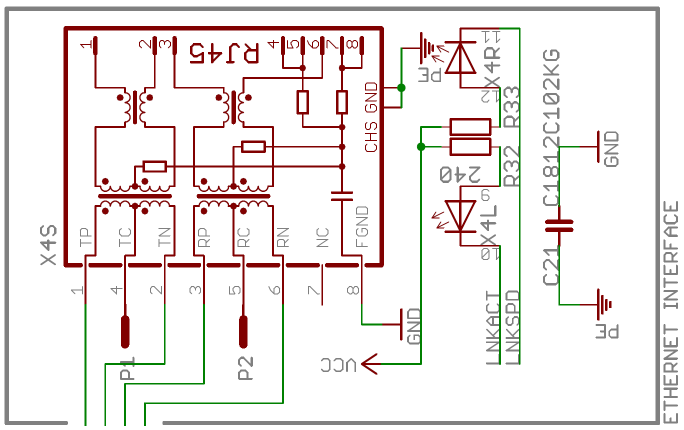



Abbildung 4.9: Schaltplan der Ethernetbuchse des Carrier-Boards[15]

## 4.3 Firmware

Zur Programmierung des Cortex-M3 stellt der hauseigene Distributor der Firma *ARM*, *KEIL*, seine Entwicklungsumgebung *µvision 5* zur Verfügung. Dieser ermöglicht die Programmierung in C und C++ und verfügt darüber hinaus über umfangreiche Debug-Funktionen und Bibliotheken, welche in neueren Programmversionen über ein spezielles Menü verwaltet und zum Projekt hinzugefügt werden können. Entsprechend der Vorgabe, wurde die Firmware in C99 verfasst.

### 4.3.1 Allgemeiner Aufbau

Die Firmware muss grundlegend vier verschiedene Aufgaben übernehmen - die Initialisierung des Systems und damit verbunden der Verschlüsselung, die Echtzeitver- und entschlüsselung des Informationssignals, die Echtzeiterkennung des Synchronisationsimpulses, sowie den Empfang des Kryptographieschlüssels. Dazu wurde die Firmware als Automat entworfen und folgt dem Schema eines endlichem Automaten mit drei Zuständen. Die Zustände sind wie folgt definiert:

- Initialisierung - Initialisierung der Hard- und Software
- Betriebsmodus - Übertragung des Signals, dabei Ver- und Entschlüsselung
- Konfigurationsmodus - Empfang und Speicherung, sowie Versand des Kryptographieschlüssels

Abbildung 4.10 zeigt das Automatenmodell der Firmware. Zu sehen sind die Übergangsmöglichkeiten zwischen den Zuständen, sowie die Zustände selbst. Zu Beginn Durchläuft der Automat den Zustand der Initialisierung $(S_0)$, anschließend verweilt er

[15] [6, S. 11, Abbildung 1]

im Betriebsmodus ($S_1$) bis er durch Änderung des nötigen Eingangs in den Konfigurationsmodus ($S_2$) übergeht. In diesem Zustand können Kryptographieschlüssel mit dem Geräte ausgetauscht werden. Bei erneuter Betätigung des Tasters, wechselt der Automat in den Betriebsmodus zurück.

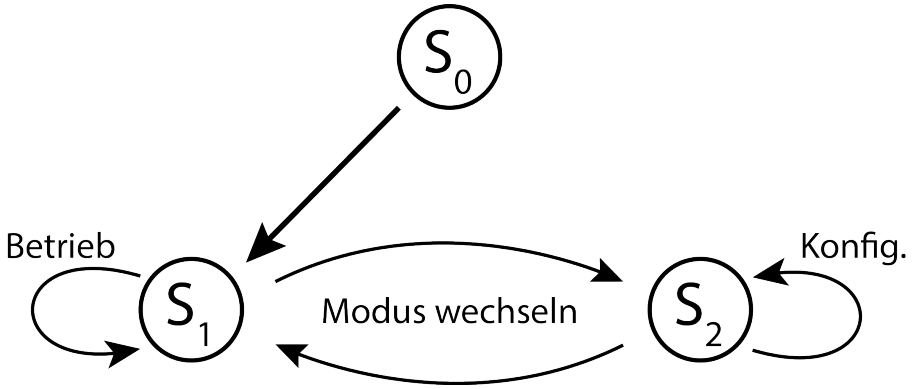



Abbildung 4.10: Schematische Darstellung des verwendeten Softwareautomaten

Im Betriebsmodus werden die Daten eingelesen, ausgewertet und ausgegeben. Im Falle des Senders wird den Daten der Synchronisationsimpuls hinzugefügt, welcher vom Empfänger erkannt werden muss um die Daten entschlüsseln zu können. Beim Empfänger hingegen besteht die Auswertung dabei aus der Filterung mit einem digitalen FIR-Filter zur Detektion des Synchronisationssignals. Der Filter ist auf dessen Frequenz abgestimmt. Wird die Präsenz eines Signals am Ausgang des Filters detektiert, so kann davon ausgegangen werden, dass diese durch die Synchronisation hervorgerufen wird. Um fehlerhafte Erkennungen zu vermeiden, und Störsignale auf gleicher Frequenz vom gewünschten Impuls differenzieren zu können, wurden einige Mechanismen etabliert. Auf diese wird später in Abschnitt 4.3.4 noch eingegangen werden. Wird der Impuls als solcher erkannt, wird der Empfänger zurückgesetzt und die Entschlüsselung der empfangenen Daten beginnt. Die Synchronisation stellt für diesen in jedem Fall den Beginn eines neuen Übertragungs- und somit Verschlüsselungsblocks dar.

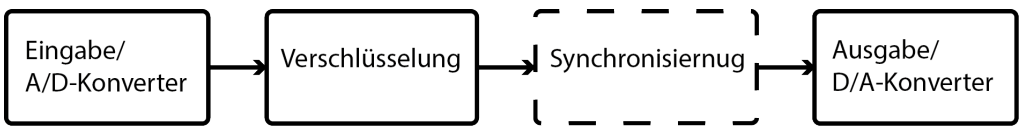



Abbildung 4.11: Block-Diagramm der Firmware des Senders

Die Abbildungen 4.11 und 4.12 zeigen den Ablauf der jeweiligen Firmware schematisch.

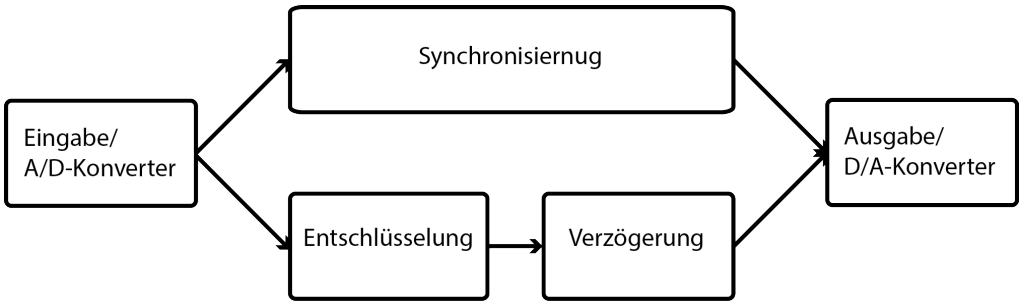



Abbildung 4.12: Block-Diagramm der Firmware des Empfängers

### 4.3.2 Bibliotheken

Moderne Mikrocontroller verwenden eine Vielzahl von zusätzlicher On-Chip-Peripherie und erlauben darüber hinaus den Anschluss zahlreicher externer Hardware. Die Kommunikation und Bedienung muss in den meisten Fällen mit Hilfe von Software realisiert werden. Da der Zugriff auf die interne und externe Hardware über Speicherregister erfolgt, wird der Quellcode oft schnell zu lang und zu unübersichtlich. Aus diesem Grund verwendet man Softwarebibliotheken. In diesen werden häufig verwendete oder sehr komplizierte Unterprogramme hinterlegt, welche im Programm selber durch eine einzelne Funktion aufgerufen werden können. Die Verwendung von Bibliotheken verbessert nicht nur die Lesbarkeit des Quellcodes, sie erhöht auch die Wiederverwendbarkeit. Anstatt ganzer Codeblöcke, welche sich nur minimal unterscheiden, können parametrierbare Funktionen verwendet werden, welche das gleiche Ergebnis liefern.
Sehr deutlich wird dieses Prinzip bei der Ansteuerung eines Displays. Anstatt für jedes auszugebende Zeichen die komplette Routine zur Darstellung des Zeichens zu verwenden, kann eine Funktion erstellt werden, welche das auszugebende Zeichen als Parameter übergeben bekommt. Die Funktion selbst führt immer den gleichen Programmcode zur Übermittlung des Zeichens aus und verwendet für das Zeichen selber den über eine Variable übergebenen Parameter.
Im Folgenden Unterabschnitt sollen einige teilweise selber erstellte Bibliotheken, die in diesem Projekt verwendet worden sind, näher betrachtet werden.

#### Timer

Der interne Timer des LPC1768 besitzt vier Kanäle, von denen jeweils zwei 12 Bit und die anderen zwei 10 Bit große Zählregister besitzen. Sie können als Timer oder Counter betrieben werden und sind in der Lage über die Capture- und Match-Funktionen auf externe Signale zu reagieren oder diese zu erzeugen. Dabei wird im Capture Modus ein Abbild des aktuellen Zählregisters erzeugt, sobald eine festgelegte Flankenänderung an einem definiertem Eingang erkannt wird. Im Match-Modus kann der Pegel eines Portpins geändert werden, sobald das Zählregister mit einem gewünschtem (Match-) Wert übereinstimmt. Zusätzlich können Interrupts für verschiedene Ereignisse Konfiguriert

```
18 void timer1_init(void) {
19   LPC_TIM1->MR0 = 1330;                    // 8000 Hz = SampleRate
20   LPC_TIM1->MCR = 7;                       // MR0
21
22   NVIC_SetPriority(TIMER1_IRQn,15);
23   NVIC_EnableIRQ(TIMER1_IRQn);
24   LPC_TIM1->TCR |= (1<<1);                 // Reset Timer0
25   LPC_TIM1->TCR &= ~(1<<1);                // Start Timer0 (clear bit)
26   LPC_TIM1->TCR |= (1<<0);                 // Enable Timer0
27 }
```

Quellcode 4.1: Bibliothek des internen Timers

werden.
Für die Verwendung als Zeitbasis des Systems wird lediglich der Timer-Modus benötigt. Zur Konfiguration besitzt die Bibliothek hierfür einzelne Funktionen, die vor der Verwendung angepasst werden müssen. Da der Timer nur ein einziges mal zu Beginn des Programms initialisiert wird, wurde auf Funktionsparameter verzichtet.

Quellcode 4.1 zeigt die für die Initialisierung der Timer zuständige Funktion. Diese existiert für jeden Timer jeweils einmal. Zu Beginn wird das Matchregister mit dem gewünschten Wert, bei dem ein Interrupt ausgelöst werden soll beschrieben. Im vorliegenden Fall handelt es sich dabei um den Wert $1330$. Der Timer soll die Zeitbasis für die Abtastung des Audiosignals darstellen. Bei einer Taktfrequenz von $25\mathrm{MHz}$ und einer gewünschten Abtastrate von $16\mathrm{kHz}$ ergibt sich theoretisch ein Matchwert von $1563$ (Gleichung 4.1)[16]. Durch die Programmlaufzeit der Interrupt-Service-Routine (*ISR*) und den Sprungvorgang in diese, verringert sich der Wert allerdings auf $1330$ (Quellcode 4.1, Zeile 19). Da der Empfänger im Gegensatz zum Sender mit einer höheren Taktfrequenz von $120\mathrm{MHz}$ arbeitet, ist der Matchwert bei diesem entsprechend größer. Diese Werte wurde empirisch durch Zeitmessungen mit der IDE und einem Oszilloskop ermittelt. Die IDE nutzt dafür die Anzahl der Zyklen die zur Ausführung des Maschinencodes nötig sind, sowie die Taktfrequenz (Abbildung 4.13). Die Grundfrequenz des Timers beim Empfänger beträgt $30\mathrm{MHz}$. Für eine Abtastperiodendauer von $62.5\mu\mathrm{s}$ werden demzufolge 1875 Zyklen benötigt, abzüglich der Programmlaufzeit ergibt dies 1862 Takte. Zur Überprüfung der Werte mit dem Oszilloskop wurde ein Pegelwechsel an einem digitalen Ausgang erzeugt, sobald die ISR aufgerufen wird. Die Zeitdifferenz zwischen den Flanken entspricht der Laufzeit des Programms (Abbildung 4.14).

[16] [17, S. 490]

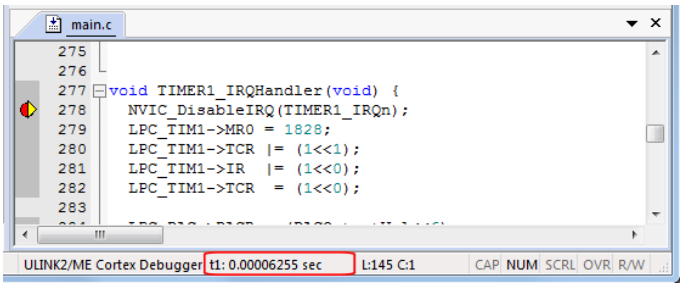



Abbildung 4.13: Zeitmessung mittels IDE

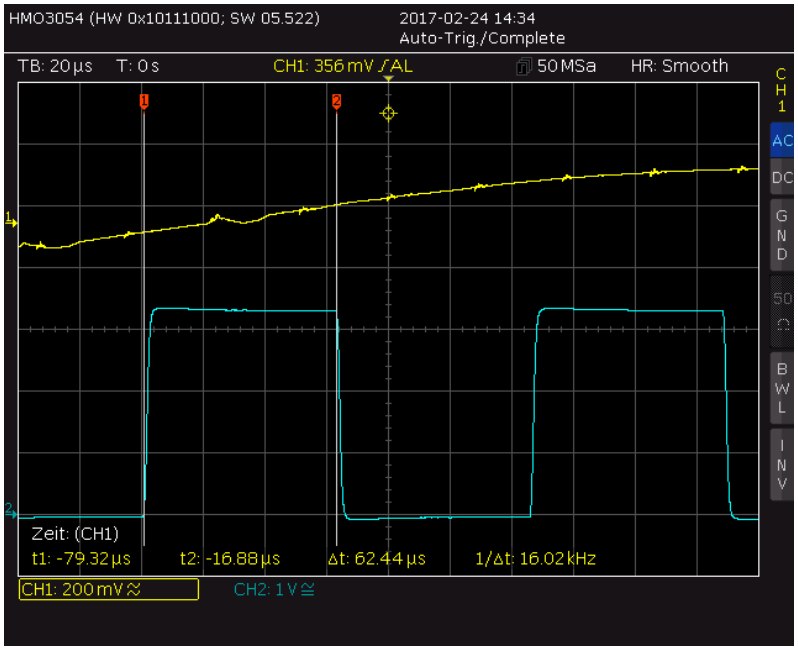



Abbildung 4.14: Dauer des eingestellten Timers

Zusätzlich zum verwendeten Timermodus, wird die Matchfunktion des Controllers genutzt. Diese erlaubt das auslösen eines Interrupts, sobald der festgelegte Matchwert erreicht wurde. Dazu müssen die ersten drei Bits des *Match Control Registers* gesetzt werden (Quellcode 4.1, Zeile 24). Dies führt dazu, dass der Interrupt des Timer1 ausgelöst wird, sobald der Zähler den Wert des *Match Registers* erreicht hat. Anschließend wird der Timer zurückgesetzt und angehalten.
Um sicherzustellen, dass die Zeitbasis unter allen Umständen zur Verfügung steht, wird die Priorität des Interrupts erhöht, somit wird die ISR des Timers nicht durch andere Interrupts unterbrochen (Quellcode 4.1, Zeile 26). Bevor der Timer abschließend gestartet werden kann, wird der Interrupt aktiviert und das Zählregister zurückgesetzt (Quellcode 4.1, Zeile 27 ff.).

$$M = \frac{f_s}{f_{clk}} = \frac{16\text{kHz}}{25\text{MHz}} = 1563 \tag{4.1}$$

### Synchronous Serial Port

Der LPC1768 ermöglicht die Kommunikation mit externer Hardware über das SPI-Protokoll (serial peripherial interface). Dazu besitzt der Mikrocontroller einen so genannten *Synchronous Serial Port*. Dieser unterstützt sowohl die Kommunikation über SPI, als auch SSI und Microwire.

#### *spi_init()*

Da der SSP mehrere Protokolle unterstützt, ist eine Konfiguration des Controllers vor der Verwendung nötig. Neben den Ein- und Ausgängen für die Steuer- und Datenleitungen des Busses (Quellcode 4.2, Zeile 7 ff.), müssen auch die Taktfrequenz (Quellcode 4.2, Zeile 15 und 18), das Format des Datenframes (Quellcode 4.2, Zeile 16 f.) sowie der Interrupt konfiguriert werden (Quellcode 4.2, Zeile 11 f.).
Der Controller kommuniziert mit Hilfe eines 13 Bit Datenrahmens, wobei 3 Takte zur Synchronisierung verwendet und anschließend 10 Bit Nutzdaten übertragen werden. Da die Bibliothek ausschließlich zur Kommunikation mit dem externen D/A-Wandler *MCP3001* verwendet wird, richtet sich das Format des Datenframes allein nach dessen Vorgaben. Wie in Abbildung 4.15 zu sehen, wird das erste Nutzdatenbit drei Takte nach Änderung des Steuersignals *CS* übertragen.
Die Taktfrequenz ergibt sich mit den Vorteilern $\text{CPSR} = 2$ und $\text{SCR} = 4$ und dem Grundtakt der Peripherie von $25\text{MHz}$ nach Gleichung 4.2 zu $2.5\text{MHz}$ [17, S. 422, Tabelle 370].

```
5  void spi_init(void) {
6    // SSP in SPI-MODE
7    LPC_PINCON->PINSEL0 |= (2UL<<30); // Set GPIO to SCLK
8    LPC_PINCON->PINSEL1 |= (10UL<<0); // Set SSEL,MISO
9    LPC_GPIO0->FIODIR |= (7<<15);     // Output
10
11   LPC_SSP0->IMSC |= (1<<0);         // Enable IRQ on FIFO overflow (Rx)
12                                     // or completly received Frame
13                                     // Enable IRQ on TimeOut(Rx)
14   NVIC_EnableIRQ(SSP0_IRQn);        // Enable global interrupts for SSP0
15
16   // Clock = PCLK / ( CPSR * (SCR+1) ) == 25Mhz/ (2*(4+1)) = 2.5Mhz
17   LPC_SSP0->CPSR = 2;               // Prescale Minimum 2
18   LPC_SSP0->CR0 |= (12<<0)          // 13bit Frame (10bit Data, 3 Bit Sync)
19                  | (0<<7)           // CPHA 1
20                  | (4<<8);          // Prescale to 5
```

Quellcode 4.2: Bibliothek zur Ansteuerung der SPI-Schnitstelle (1)

```
25 int RecSPI (void) {
26   int data;
27   /* Receive SPI data to buffer. */
28     LPC_SSP0->DR = 0x3FF;
29     /* Wait while Rx FIFO is empty. */
30     while (!(LPC_SSP0->SR & RNE));
31     data = (LPC_SSP0->DR)&0x3FF;
32     return data;
33 }
```

Quellcode 4.3: Bibliothek zur Ansteuerung der SPI-Schnitstelle (2)

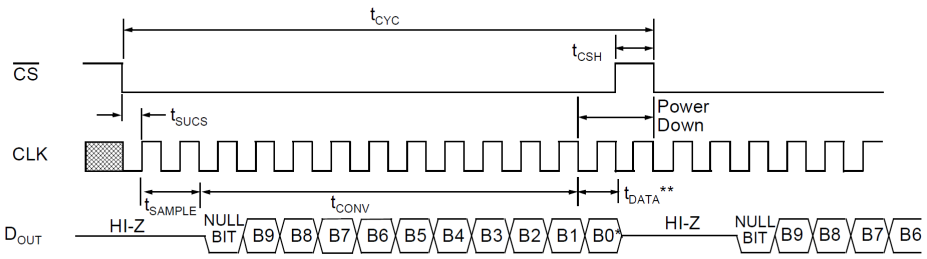



Abbildung 4.15: Format des vom A/D-Wandler übertragenem Datenframes[17]

$$f_{clk} = \frac{\text{PCLK}}{\text{CPSR} \cdot (\text{SCR}+1)} = \frac{25\text{MHz}}{2 \cdot 5} = 2.5\text{MHz} \tag{4.2}$$

***RecSPI()***

Für den Empfang von Daten, die von den an den Bus angeschlossenen Slaves übermittelt werden, enthält die Bibliothek die Funktion *RecSPI()*. Die Übermittlung wird gestartet, indem das *SSP Data Register* beschrieben wird (Quellcode 4.3, Zeile 28). Anschließend ändert der Controller den Pegel der Steuerleitung *CS*, auch *SSEL* genannt, auf *low* und der Slave beginnt mit der Übertragung seiner Daten. Sobald der Frame vollständig Übertragen und das RNE-Bit (*Receive FIFO not empty*) gesetzt ist, werden die Daten aus dem *SSP Data Register* ausgelesen und als Rückgabewert der Funktion ausgegeben (Quellcode 4.3, Zeile 30 ff.).

### Display

Für das verwendete Display vom Typ *GDM2004D* des Herstellers *XMOCULAR* wurde eine umfangreiche Bibliothek zur Ansteuerung des verbauten Controllers *ST7066U-0A* erstellt. Einige der darin enthaltenen und in Tabelle 4.1 aufgeführten Funktionen sollen

[17] [16, S. 15, Abbildung 5-1]

hier näher erläutert werden. Die Vollständige Bibliothek ist in Anhang C zu finden.

| Funktion | Bemerkung |
|---|---|
| lcd_command() | Übertragung von Befehlen an den Displaycontrollers |
| lcd_init() | Initialisierung des Displays |
| lcd_clear() | Löschen des Bildspeichers |
| lcd_cursor() | Positionierung des Cursors |
| lcd_putchar() | Schreiben eines einzelnen Characters |
| lcd_putstring() | Schreiben einer Zeichenkette |

Tabelle 4.1: Funktionsübersicht der Bibliothek lcd_4bit_hd44780.c

***lcd_command()***

Der an das Display angeschlossene Controller wird parallel über Portpins gesteuert. Dazu verfügt dieser über drei Steuerleitungen (enable, register select und read/write select) und vier oder acht Datenleitungen. Mit Hilfe der Steuerleitungen wird ausgehandelt, ob Befehle oder Daten übertragen oder Daten aus dem Speicher des Controllers ausgelesen werden sollen. Darüber hinaus signalisieren sie, dass die Pegel der Datenleitungen gesetzt wurden und der Befehl vom Controller eingelesen und verarbeitet werden kann (Quellcode 4.4, Zeile 19 ff.).
Um die Anzahl der benötigten Datenleitungen zu verringern, kann der Controller im 4-Bit-Modus betrieben werden. Dazu verwendet er nur vier Portpins, über die in zwei Schritten jeweils ein Halbbit des 8-Bit langen Befehls übertragen werden. Hierfür werden zuerst die vier höherwertigen Bits und anschließend die vier niederwertigen Bits an den Portpins angelegt (Quellcode 4.4 Zeile 24 und 34). Im, hier nicht verwendeten, 8-Bit-Modus wird der Befehl in einem Durchlauf übertragen. Eine Übersicht über den Befehlssatz des Controllers und die dazugehörigen Bitcodes gibt Abbildung 4.16.

***lcd_init()***

Die in Abbildung 4.16 aufgeführten Befehle erlauben eine umfangreiche Konfiguration des Controllers und des angeschlossenen Displays. Diese wird mit der Funktion *lcd_init()* realisiert. Dazu werden die einzelnen Bitcodes mit Hilfe des Unterprogramms *lcd_command()* übertragen.

Zuerst müssen die benötigten Datenleitungen als Ausgänge definiert werden (Quellcode 4.5, Zeile 44 f.). Im Anschluss werden die Befehle zur Konfiguration des Display übertragen. Das Display wird im zweizeiligen 4-Bit-Modus mit einer Zeichengröße von 5x11 Bildpunkten betrieben. Außerdem wird die Darstellung des Cursors und die Art seiner Bewegung konfiguriert. Abschließend wird der Bildspeicher des Controllers ge-

```
18 void lcd_command(int cmd) {
19   LPC_GPIO2->FIOSET |= (1<<2);                    //EN high
20   LPC_GPIO2->FIOCLR |= (1<<0);                    //RS low (command mode)
21   LPC_GPIO2->FIOCLR |= (1<<1);                    //RW low (write mode)
22   LPC_GPIO2->FIOCLR |= (3<<3);                    //Clear Dataport (P2.3&P2.4)
23   LPC_GPIO1->FIOCLR |= (3UL<<30);                 //Clear Dataport (P1.31&P1.31)
24   LPC_GPIO2->FIOSET |= (((cmd>>4)&0x03)<<3);      //Set Dataport to
25                                                   //higher nibble D4/D5
26   LPC_GPIO1->FIOSET |= (((cmd>>4)&0x04)<<29);     //Set D6
27   LPC_GPIO1->FIOSET |= (((cmd>>4)&0x08)<<27);     //Set D7
28   LPC_GPIO2->FIOCLR |= (1<<2);                    //disable EN, to send data
29   delay_ms(50);
30   if ( cmd != 0x20 && cmd != 0x30) {
31     LPC_GPIO2->FIOSET |= (1<<2);                  //EN high
32     LPC_GPIO2->FIOCLR |= (3<<3);                  //Clear Dataport (P2.3&P2.4)
33     LPC_GPIO1->FIOCLR |= (3UL<<30);               //Clear Dataport (P1.31&P1.31)
34     LPC_GPIO2->FIOSET |= ((cmd&0x03)<<3);         //Set Dataport to
35                                                   //lower nibble D4/D5
36     LPC_GPIO1->FIOSET |= ((cmd&0x04)<<29);        //Set D6
37     LPC_GPIO1->FIOSET |= ((cmd&0x08)<<27);        //Set D7
38     LPC_GPIO2->FIOCLR |= (1<<2);                  //disable EN, to send data
39   }
```

Quellcode 4.4: Bibliothek für die Verwendung des LC-Displays - Funktion lcd_command()

löscht, um das Display zu leeren (Quellcode 4.5, Zeile 55 ff.) Die genauen Bedeutungen der numerischen Befehlscodes können der Übersicht in Abbildung 4.16 entnommen werden.

### *lcd_clear()*

Zum Löschen des Bildinhaltes kann der Befehl *lcd_clear()* verwendet werden. Dieser übermittelt eine einzelne Anweisung an den Controller, welche dazu führt, dass der Bildspeicher geleert wird. Anschließend kann dieser mit neuem Inhalt beschrieben werden (Quellcode 4.6).

### *lcd_cursor()*

Der Controller verwendet einen Cursor zur Orientierung auf dem Display. Dazu werden die Zellen in eine Matrix eingeteilt, welche in der oberen, linken Ecke mit den Koordinaten $(0,0)$ beginnt. Alle Schreib- und Löschoperationen werden an der aktuellen Position des Cursors durchgeführt. Die Funktion *lcd_cursor()* erlaubt das setzen des Zeigers auf eine beliebige Stelle, dazu müssen die Koordinaten als Parameter übergeben werden (Quellcode 4.7). Intern werden die Koordinaten durch die Speicheradressen repräsentiert, welche den Inhalt für das jeweilige Feld des Displays inne hat. Daher ist eine Konvertierung der Koordinaten in die entsprechenden Adressen nötig. Abbildung 4.17 zeigt die Speicheraufteilung des Displaycontrollers.

| | RS | R/W | DB7 | DB6 | DB5 | DB4 | DB3 | DB2 | DB1 | DB0 | | time (fosc= 270 KHZ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Clear Display | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Write “20H” to DDRA and set DDRAM address to “00H” from AC | 1.53ms |
| Return Home | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | - | Set DDRAM address to “00H” From AC and return cursor to Its original position if shifted. The contents of DDRAM are not changed. | 1.53ms |
| Entry mode Set | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | I/D | SH | Assign cursor moving direction And blinking of entire display | 39us |
| Display ON/ OFF control | 0 | 0 | 0 | 0 | 0 | 0 | 1 | D | C | B | Set display (D), cursor (C), and Blinking of cursor (B) on/off Control bit. | |
| Cursor or Display shift | 0 | 0 | 0 | 0 | 0 | 1 | S/C | R/L | - | - | Set cursor moving and display Shift control bit, and the Direction, without changing of DDRAM data. | 39us |
| Function set | 0 | 0 | 0 | 0 | 1 | DL | N | F | - | - | Set interface data length (DL: 8- Bit/4-bit), numbers of display Line (N: =2-line/1-line) and, Display font type (F: 5x11/5x8) | 39us |
| Set CGRAM Address | 0 | 0 | 0 | 1 | AC5 | AC4 | AC3 | AC2 | AC1 | AC0 | Set CGRAM address in address Counter. | 39us |
| Set DDRAM Address | 0 | 0 | 1 | AC6 | AC5 | AC4 | AC3 | AC2 | AC1 | AC0 | Set DDRAM address in address Counter. | 39us |
| Read busy Flag and Address | 0 | 1 | BF | AC6 | AC5 | AC4 | AC3 | AC2 | AC1 | AC0 | Whether during internal Operation or not can be known By reading BF. The contents of Address counter can also be read. | 0us |
| Write data to Address | 1 | 0 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | Write data into internal RAM (DDRAM/CGRAM). | 43us |
| Read data From RAM | 1 | 1 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | Read data from internal RAM (DDRAM/CGRAM). | 43us |

Abbildung 4.16: Befehlssatz des Controllers ST7066U-0A

```
43 void lcd_init(void) {
44   LPC_GPIO2->FIODIR |= (31<<0);                    // Define Outputpins
45   LPC_GPIO1->FIODIR |= (3UL<<30);
46   delay_ms(200);
47   lcd_command(0x30);
48   lcd_command(0x30);
49   lcd_command(0x30);
50   lcd_command(0x20);
51   lcd_command(0x2C);
52   lcd_command(0x14);
53   lcd_command(0x06);
54   lcd_command(0x0C);
55   lcd_command(0x01);
56 }
```

Quellcode 4.5: Bibliothek für die Verwendung des LC-Displays - Funktion lcd_init()

```
120 void lcd_clear(void) {
121   lcd_command(0x01);
122 }
```

Quellcode 4.6: Bibliothek für die Verwendung des LC-Displays - Funktion lcd_clear()

DDRAM address:

Display position

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F | 10 | 11 | 12 | 13 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 4A | 4B | 4C | 4D | 4E | 4F | 50 | 51 | 52 | 53 |
| 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D | 1E | 1F | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 54 | 55 | 56 | 57 | 58 | 59 | 5A | 5B | 5C | 5D | 5E | 5F | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 |

DDRAM address

Abbildung 4.17: Speicheraufteilung des Controllers ST7066U-0A

```
89  void lcd_cursor(int x,int y) {
90    LPC_GPIO2->FIOSET |= (1<<2);                    //EN high
91    LPC_GPIO2->FIOCLR |= (1<<0);                    //RS low (command mode)
92    LPC_GPIO2->FIOCLR |= (1<<1);                    //RW low (write mode)
93    switch(y) {
94      case 1:
95        y=0x80;
96        break;
97      case 2:
98        y=0xC0;
99        break;
100     case 3:
101       y=0x94;
102       break;
103     case 4:
104       y=0xD4;
105       break;
106     default:
107       y=0x80;
108       break;
109   }
110   if ( x > 20 ) { x = 20; }
111   y+=x;
112   lcd_command(y);
113 }
```

Quellcode 4.7: Bibliothek für die Verwendung des LC-Displays - Funktion lcd_cursor()

```
82 void lcd_putstring(unsigned char *s) {
83   while (*s != '\0') {
84     lcd_putchar((*s));
85     s++;
86   }
87 }
```

Quellcode 4.8: Bibliothek für die Verwendung des LC-Displays - Funktion lcd_putstring()

```
58 void lcd_putchar(unsigned char c) {
59   LPC_GPIO2->FIOSET |= (1<<2);                  //EN high
60   LPC_GPIO2->FIOSET |= (1<<0);                  //Charactermode (RS High)
61   LPC_GPIO2->FIOCLR |= (1<<1);                  //RW low (write mode)
62   LPC_GPIO2->FIOCLR |= (3<<3);                  //Clear Dataport (P2.3&P2.4)
63   LPC_GPIO1->FIOCLR |= (3UL<<30);               //Clear Dataport (P1.31&P1.31)
64
65   LPC_GPIO2->FIOSET |= (((c>>4)&0x03)<<3);      //Set Dataport to
66                                                 //higher nibble D4/D5
67   LPC_GPIO1->FIOSET |= (((c>>4)&0x04)<<29);     //Set D6
68   LPC_GPIO1->FIOSET |= (((c>>4)&0x08)<<27);     //Set D7
69   LPC_GPIO2->FIOCLR |= (1<<2);                  //disable EN, to send data
70   delay_ms(1);
71   LPC_GPIO2->FIOSET |= (1<<2);                  //EN high
72   LPC_GPIO2->FIOCLR |= (3<<3);                  //Clear Dataport (P2.3&P2.4)
73   LPC_GPIO1->FIOCLR |= (3UL<<30);               //Clear Dataport (P1.31&P1.31)
74   LPC_GPIO2->FIOSET |= ((c&0x03)<<3);           //Set Dataport to
75                                                 //lower nibble D4/D5
76   LPC_GPIO1->FIOSET |= ((c&0x04)<<29);          //Set D6
77   LPC_GPIO1->FIOSET |= ((c&0x08)<<27);          //Set D7
78   LPC_GPIO2->FIOCLR |= (1<<2);                  //disable EN, to send data
79   delay_ms(1);
80 }
```

Quellcode 4.9: Bibliothek für die Verwendung des LC-Displays - Funktion lcd_putchar()

***lcd_putchar() und lcd_putstring()***

Zur Darstellung von Zeichen wurden die Funktionen *lcd_putchar()* und *lcd_putstring()* entwickelt. Die Anweisung zur Übertragung eines einzelnen Zeichens, *lcd_putchar()*, überträgt dabei den, zum übergebenen Character äquivalenten, Zeichencode (Quellcode 4.9). Zeichenketten werden dagegen mit *lcd_putstring()* dargestellt. Der Befehl überträgt dabei jedes Zeichen des Strings einzeln, bis das Abschlusszeichen $\backslash 0$ der Zeichenkette erreicht wurde (Quellcode 4.8, Zeile 83 ff.).

### D/A-Wandler

Zur Ausgabe des analogen Signals, wird der interne A/D-Wandler des LPC1768 verwendet. Die Ansteuerung des Wandlers erfolgt über die entsprechenden Register des Mikrocontrollers. Die Ausgabe analoger Spannungen besteht nur aus einem einzigen Befehl und benötigt daher keine eigene Funktion. Aus diesem Grund enthält die Bibliothek für den D/A-Wandler lediglich das Unterprogramm *init_dac()* zur Initialisierung des

Controllers.

```
4 void init_dac(void) {
5     LPC_GPIO0->FIODIR    |= (1<<26);          // P0.26 Output ( P18 DAC )
6     LPC_PINCON->PINMODE1 |= (1<<21);          // dac no pullup/down
7     LPC_PINCON->PINSEL1  |= (1<<21);          // P0.26 = DAC
8 }
```

Quellcode 4.10: Bibliothek für den D/A-Wandler des LPC1768

Der Ausgang des DAC ist eine von mehreren Mehrfachbelegungen eines Portpins. Damit die Spannung auf diesem ausgegeben wird, muss die entsprechende Funktion im *Pin function select register* ausgewählt werden (Quellcode 4.10, Zeile 7). Außerdem werden die Pullup/-down Wiederstände deaktiviert, und der Portpin aus Ausgang deklariert (Quellcode 4.10, Zeile 5 f.).
Mit dem Aufruf dieser Funktion ist der Wandler funktionsbereit und die gewünschten Digitalwerte können ausgegeben werden. Dazu muss das *D/A Converter Register* (*DACR*) mit dem entsprechendem Wert beschrieben werden. Zu beachten ist, dass das niederwertigste Bit bei Position sechs beginnt. Der Befehl würde demnach wie folgt aussehen:

```
1   LPC_DAC->DACR = (512<<6);
2
```

### Netzwerkstack

Zur Kommunikation über die Netzwerkschnittstelle und Ansteuerung des *EMAC* und des *PHY* stellt *KEIL* eine Bibliothek für den *LPC1768* und das dazugehörige Entwicklungsboard *MCB1700* zur Verfügung. Diese ist vollständig kompatibel mit dem *Chip1768*, sowie dem Carrier-Board *TestBed* und bietet eine Vielzahl von Zusatzfunktionen. So dient sie nicht nur der Kommunikation über die Ethernetschnittstelle, sie Implementiert Dienste und Funktionen, die im OSI-Referenzmodell bis auf die oberste Schicht der Anwendungsebene reichen. Neben Transportprotokollen wie TCP/IP und UDP stehen darin umfangreiche Serverdienste für HTTP und FTP zur Verfügung. Um diese Features nutzen zu können, muss allerdings eine gültige *MDK-Pro-Lizenz* für die Entwicklungsumgebung $\mu$*Vision* vorliegen, da die verschlüsselte Bibliothek sich nur mit dieser kompilieren lässt.

Eine *leightweight*, also eine *„leichtgewichtige“*, Alternative stellt der Stack von *Andreas Dannenberg*, der 2001 an der *HTWK Leipzig* für den *MSP430* von *Texas Instruments* veröffentlich wurde, dar. Er bietet die grundlegenden Fähigkeiten zur Kommunikation mittels TCP/IP und zur Übermittlung von Inhalten via HTTP und ist im Gegensatz zur propritären Lösung von *KEIL* frei verfügbar.

Da diese Bibliothek wesentlich weniger ROM und RAM benötigt und ohne proprietäre Software von *KEIL* kompilierbar ist, findet diese Verwendung in diesem Projekt. Einige wichtige, in Tabelle 4.2 aufgeführte, Funktionen der Bibliothek sollen hier kurz näher gebracht werden. Tatsächlich umfasst sie weit mehr Funktionen, welche von den hier genannten genutzt werden. Da diese jedoch nicht zur direkten Implementierung des *API* (*application programming interface*) benötigt werden, wird auf eine Erklärung verzichtet. Der gesamte Quellcode der Bibliothek kann in Anlage C eingesehen werden.

| Funktion | Bemerkung |
|---|---|
| TCPLowLevelInit() | Initialisierung der Soft- und Hardware |
| TCPPassiveOpen() | Starten eines Listeners auf einem Socket |
| TCPReleaseRxBuffer() | Freigeben des Empfangsspeichers |
| TCPTransmitTxBuffer() | Übermittlung des Sendespeichers |
| DoNetworkStuff() | zyklische Abarbeitung der State-Machine |
| TCPClose() | Abbau einer aktiven TCP-Verbindung |

Tabelle 4.2: Funktionsübersicht der Bibliothek „easyweb“ von Andreas Dannenberg

**TCPLowLevelInit()**

Bevor die Netzwerkschnitstelle genutzt werden kann, erfolgt die Initialisierung der Hard- und Software mittels der Funktion *TCPLowLevelInit()*.

```
126 void TCPLowLevelInit(void)
127 {
128   Start_SysTick10ms();   // Start SysTick timer running (10ms ticks)
129
130   Init_EthMAC();
131
132   TransmitControl = 0;
133   TCPFlags = 0;
134   TCPStateMachine = CLOSED;
135   SocketStatus = 0;
136 }
```

Quellcode 4.11: Bibliothek für den Netzwerkstack (1)

Da für das Polling und Timing der Hardware eine Zeitbasis benötigt wird (zum Beispiel zum feststellen von Timeouts), wird der SysTickTimer gestartet. Dieser ist direkt an den Takt der CPU gekoppelt und arbeitet unabhängig von anderen Timern. Die Ticks bzw. Zeitschritte erfolgen dabei im Abstand von $10\text{ms}$ (Quellcode 4.11, Zeile 128). Anschließend werden mit dem Aufruf von *InitEthMAC()* sowohl der *EMAC*, als auch das *PHY* konfiguriert (Quellcode 4.11, Zeile 130). Zur Steuerung des als *state machine* umgesetzten Automaten werden einige Zustandsvariablen benötigt, welche abschließend

noch initialisiert werden (Quellcode 4.11, Zeile 132 ff.). *TCPSTateMachine* enthält dabei den Zustand des TCP-Sockets. Dieser kann insgesamt elf verschiedene, im Standard *RFC 793* festgelegte, Zustände annehmen. [9, Vgl. S. 22] *TCPFlags* entspricht den *control flags*, wie sie in [9] für die Signalisierung im TCP-Header definiert sind.
Der *SocketStatus* dient der Kontrolle des verwendeten Netzwerksockets. Er enthält den aktuellen Zustand oder einen Hinweis auf einen eventuell aufgetretenen Fehler. Mit *TransmitControl* wird unterschieden ob es sich beim zu sendenden Paket um den Handshake oder bereits um Nutzdaten handelt.

**DoNetworkStuff()**

Die Funktion *DoNetworkStuff()* beinhaltet den eigentlichen Automaten und bildet die Hauptkomponente für die Abarbeitung von Ereignissen der Netzwerkschnittstelle. Sie dient den Empfang, der Verarbeitung und dem Versand von Daten.

```
253 void DoNetworkStuff(void)
254 {
255   // Check to see if packet received
256   if (CheckIfFrameReceived())
257   {
258   // Was it a broadcast message?
259     if (BroadcastMessage()) {
260       ProcessEthBroadcastFrame();
261     }
262     else {
263       ProcessEthIAFrame();
264     }
265     // now release ethernet controller buffer
266     StopReadingFrame();
267   }
```

Quellcode 4.12: Bibliothek für den Netzwerkstack (2)

Im ersten Schritt wird geprüft ob auf dem Socket ein Datenframe empfangen wurde und ob es sich dabei um einen Broadcast handelt oder nicht. Im Falle eines ARP-Broadcast (*address resolution protocol*) wird die Übermittlung der eigenen logischen und physischen Adresse als Antwort darauf vorbereitet, ansonsten wird das Paket verworfen (Quellcode 4.12, Zeile 256 ff.). Der eigentliche Versand der Antwort findet erst am Ende der Prozedur statt.
Handelt es sich bei dem Paket nicht um einen Broadcast, wird das Paket nach seinem Inhalt untersucht. Die Bibliothek erlaubt dabei die Erkennung von ARP-Antworten zur Ermittlung der Hardware-Adresse von Kommunikationspartnern, sowie die Verarbeitung von IP- (*Internet Protocol*) und ICMP-Paketen (*Internet Control Message Protocol*). Handelt es sich um ein reines IP-Paket, das kein ICMP-Paket darstellt, wird geprüft ob es sich um ein Paket der TCP-Protokollfamilie handelt. In allen Fällen wird wiederum die entsprechende Antwort vorbereitet, jedoch noch nicht gesendet. Dabei kann es sich

um einen Verbindungsaufbau oder -abbau, sowie Nutzdaten für eine bereits bestehende Verbindung handeln (Quellcode 4.12, Zeile 263).
Nachdem der aktuelle Frame bearbeitet wurde, wird der Zwischenspeicher für den erneuten Empfang geleert (Quellcode 4.12, Zeile 266).

```
271   if (TCPFlags & TCP_TIMER_RUNNING)
272     if (TCPFlags & TIMER_TYPE_RETRY)
273     {
274       if (TCPTimer > RETRY_TIMEOUT)
275       {
276         TCPRestartTimer();                    // set a new timeout
277
278         if (RetryCounter)
279         {
280           TCPHandleRetransmission();          // resend last frame
281           RetryCounter--;
282         }
283         else
284         {
285           TCPStopTimer();
286           TCPHandleTimeout();
287         }
288       }
289     }
290     else if (TCPTimer > FIN_TIMEOUT)
291     {
292       TCPStateMachine = CLOSED;
293       TCPFlags = 0;                           // reset all flags, stop
    retransmission...
294       SocketStatus &= SOCK_DATA_AVAILABLE;// clear all flags but data
    available
295     }
```

Quellcode 4.13: Bibliothek für den Netzwerkstack (3)

Wie bereits erwähnt dient die Funktion jedoch nicht nur der Verarbeitung der Daten, sondern auch der Verwaltung des Sockets und der TCP-Verbindung. Aus diesem Grund wird im nächsten Schritt geprüft ob ein Timer gestartet wurde um eine Zeitüberschreitung bei der Übertragung von Paketen zu erkennen. Ist dies der Fall und bereits eine Zeitüberschreitung eingetreten, so wird das Paket nochmals übertragen, sofern die Höchstanzahl an versuchen noch nicht erreicht wurde (Quellcode 4.13, Zeile 274 ff.). Ist die Übertragung hingegen bereits mehrfach fehlgeschlagen, wird die Verbindung mit einer Fehlermeldung beendet (Quellcode 4.13, Zeile 285 f.).
Tritt die Zeitüberschreitung nicht bei der Übermittlung eines Paketes auf, wird die Verbindung ohne Fehlermeldung geschlossen und zurückgesetzt, um halbgeschlossene TCP-Verbindungen zu verhindern. Die Daten bleiben dabei erhalten und können weiterhin genutzt werden (Quellcode 4.13 Zeile 290 ff.).

Nach der Überprüfung des Zustandes der TCP-Verbindung folgt der eigentliche Automat des Sockets. Er ist als switch-Anweisung organisiert. Diese erlaubt die Erkennung

```
297   switch (TCPStateMachine)
298   {
299     case CLOSED :
300     case LISTENING :
301     {
302       if (TCPFlags & TCP_ACTIVE_OPEN)              // stack has to open a
      connection?
303         if (TCPFlags & IP_ADDR_RESOLVED)           // IP resolved?
304           if (!(TransmitControl & SEND_FRAME2))    // buffer free?
305           {
306             // change TAR -> TOTC to use LPC1768 clock
307             // TCPSeqNr = ((unsigned long)ISNGenHigh << 16) | TAR;
308             TCPSeqNr = ((unsigned long)ISNGenHigh << 16) | (LPC_TIM0->TC & 0
      xFFFF); // set local ISN
309             TCPUNASeqNr = TCPSeqNr;
310             TCPAckNr = 0;                          // we don't know what to ACK!
311             TCPUNASeqNr++;                         // count SYN as a byte
312             PrepareTCP_FRAME(TCP_CODE_SYN);        // send SYN frame
313             LastFrameSent = TCP_SYN_FRAME;
314             TCPStartRetryTimer();                  // we NEED a retry-timeout
315             TCPStateMachine = SYN_SENT;
316           }
317       break;
318     }
```

Quellcode 4.14: Bibliothek für den Netzwerkstack (4)

des aktuellen Zustandes und die Abarbeitung der zugehörigen Aufgaben.
Soll aktiv eine Verbindung hergestellt werden, obwohl ein Server gestartet wurde, wird der LISTENING-Block abgearbeitet. Dazu werden zum einen die Sequenznummern für die SYN- und ACK-Flags zurückgesetzt, als auch das entsprechende Paket für eine Verbindungsanfrage an die Gegenstelle vorbereitet. Nachdem der Timeout-Timer gestartet wurde, wird der Automat in den Zustand *SYN_SENT* versetzt. Er wartet nun auf die Bestätigung des Handshakes (Quellcode 4.14, Zeile 308).

Wurde eine Verbindung erfolgreich hergestellt, wird der Block *ESTABLISHED* ausgeführt. Für eine aktive Verbindung gibt es nur eine Aktion. Sie kann lediglich aktiv oder passiv geschlossen werden. In diesem Fall wird das aktive Beenden der Verbindung behandelt (Quellcode 4.15, Zeile 322). Wird die Funktion TCPClose() aufgerufen, so wird das Flag *TCP_CLOSE_REQUESTED* gesetzt. Falls alle Daten Übertragen und bestätigt worden sind, wird der Frame für den Verbindungsabbau vorbereitet (Quellcode 4.15, Zeile 323 f). Dazu wird das *FIN*-Flag gesetzt. Anschließend wird der Zustand des Automaten auf *FIN_WAIT* geändert (Quellcode 4.15, Zeile 327 ff.). Es wird nun auf die entsprechende Antwort des Kommunikationspartners gewartet. Erfolgt eine Bestätigung, wird die Verbindung beendet.
Wird der Verbindungsabbau durch die Gegenstelle initiiert, befindet sich der Automat im Zustand *CLOSE_WAIT*. Wurden alle Daten gesendet und bestätigt, wird die Anforderung durch setzen der Flags *FIN* und *ACK* beantwortet. Die state machine begibt sich nun in den Zustand *LAST_ACK* und wartet auf eine erneute Bestätigung des Paketes.

```
319     case SYN_RECD :
320     case ESTABLISHED :
321     {
322       if (TCPFlags & TCP_CLOSE_REQUESTED)       //user has user initated a
    close?
323         if (!(TransmitControl & (SEND_FRAME2 | SEND_FRAME1)))//buffers free?
324           if (TCPSeqNr == TCPUNASeqNr)           //all data ACKed?
325           {
326             TCPUNASeqNr++;
327             PrepareTCP_FRAME(TCP_CODE_FIN | TCP_CODE_ACK);
328             LastFrameSent = TCP_FIN_FRAME;
329             TCPStartRetryTimer();
330             TCPStateMachine = FIN_WAIT_1;
331           }
332       break;
333     }
334     case CLOSE_WAIT :
335     {
336       if (!(TransmitControl & (SEND_FRAME2 | SEND_FRAME1)))//buffers free?
337         if (TCPSeqNr == TCPUNASeqNr)                 //all data ACKed?
338         {
339           TCPUNASeqNr++;                             //count FIN as a byte
340           PrepareTCP_FRAME(TCP_CODE_FIN | TCP_CODE_ACK);//we NEED a retry-
    timeout
341           LastFrameSent = TCP_FIN_FRAME;             //time to say goodbye...
342           TCPStartRetryTimer();
343           TCPStateMachine = LAST_ACK;
344         }
345       break;
346     }
347   }
```

Quellcode 4.15: Bibliothek für den Netzwerkstack (5)

Anschließend wird die Verbindung abgebaut und der Socket steht für neue Anfragen zur Verfügung.
In den Zuständen *SYN_RECD* und *CLOSED*, werden keine Befehle abgearbeitet.

```
349   if (TransmitControl & SEND_FRAME2)
350   {
351     RequestSend(TxFrame2Size);
352
353     if (Rdy4Tx())                  // NOTE: when using a very fast MCU, maybe
354       SendFrame2();                // the CS8900 isn't ready yet, include
355     else {                         // a kind of timer or counter here
356       TCPStateMachine = CLOSED;
357       SocketStatus = SOCK_ERR_ETHERNET;// indicate an error to user
358       TCPFlags = 0;                // clear all flags, stop timers etc.
359     }
360
361     TransmitControl &= ~SEND_FRAME2;    // clear tx-flag
362   }
```

Quellcode 4.16: Bibliothek für den Netzwerkstack (6)

Wurden lediglich Frames zur Übertragung vorbereitet, welche ausschließlich Metadaten in Form des Headers und keine Daten enthalten, so wurde das Flag *SEND_FRAME2* gesetzt (Quellcode 4.16, Zeile 349). Um den Speicherdirektzugriff (*DMA - direct memory access*) zu ermöglichen, wird dem Controller die Länge des zu übertragenen Frames mitgeteilt (Quellcode 4.16, Zeile 351). Anschließend erfolgt die Übertragung des Headers und das Flag wird zurückgesetzt (Quellcode 4.16, Zeile 354 ff.).

Enthält das Paket über den Header hinaus noch Nutzdaten, wurde während der Vorbereitung des Frames das Flag *SEND_FRAME1* gesetzt (Quellcode 4.17, Zeile 364). Bevor die Übertragung eingeleitet wird, muss der Header erstellt und mit den Nutzdaten vereint werden. Anschließend wird die Größe des Paketes auch hier an den DMA-Controller übergeben (Quellcode 4.17, Zeile 366 f.). Wie zuvor können die Daten nun gesendet und das entsprechende Flag gelöscht werden (Quellcode 4.17, Zeile 370 ff.). Die Abarbeitung des Zustandsautomaten für die TCP-Sitzung ist nun beendet und der Prozess kann von vorne beginnen.

```
364   if (TransmitControl & SEND_FRAME1)
365   {
366     PrepareTCP_DATA_FRAME();            // build frame w/ actual SEQ, ACK....
367     RequestSend(TxFrame1Size);
368
369     if (Rdy4Tx())                       // CS8900 ready to accept our frame?
370       SendFrame1();                     // (see note above)
371     else {
372       TCPStateMachine = CLOSED;
373       SocketStatus = SOCK_ERR_ETHERNET;// indicate an error to user
374       TCPFlags = 0;                     // clear all flags, stop timers etc.
375     }
376
377     TransmitControl &= ~SEND_FRAME1;    // clear tx-flag
378   }
379 }
```

Quellcode 4.17: Bibliothek für den Netzwerkstack (7)

**TCPPassiveOpen()**

Eingebettete System werden häufig zum Betrieb von Serveranwendungen genutzt und müssen dafür in Kommunikationsnetzen erreichbar sein. Die Bibliothek versetzt die Anwendung mit dem Befehl *TCPPassiveOpen()* in die Lage, auf einem Socket „zu lauschen“. Dies bedeutet, dass im lokalen Speicher ein Socket reserviert wird. Auf diesem werden eingehende Verbindungsanfragen zugelassen und beantwortet.

Im Falle, dass noch kein TCP-Socket geöffnet ist, wird das Flag zum passiven Öffnen, also dem Lauschen, gesetzt. Außerdem wird der Zustand des Zustandsautomaten auf *LISTENING* festgelegt. Nachdem auch der Socket durch *SOCK_ACTIVE* als aktiv gekennzeichnet wurde, wird mit der nächsten zyklischen Abarbeitung des Automaten auf

```
140 void TCPPassiveOpen(void)
141 {
142   if (TCPStateMachine == CLOSED)
143   {
144     TCPFlags &= ~TCP_ACTIVE_OPEN;                    // let's do a passive open!
145     TCPStateMachine = LISTENING;
146     SocketStatus = SOCK_ACTIVE;                      // reset, socket now active
147   }
148 }
```

Quellcode 4.18: Bibliothek für den Netzwerkstack (8)

eingehende Verbindungen auf dem, in den Variablen *MyIP* und *TCPLocalPort* definiertem, Socket gewartet (Quellcode 4.18).

**TCPClose()**

Zum Beenden von TCP-Verbindungen, kann zu jedem Zeitpunkt das Unterprogramm

```
171 void TCPClose(void)
172 {
173   switch (TCPStateMachine)
174   {
175     case LISTENING :
176     case SYN_SENT :
177     {
178       TCPStateMachine = CLOSED;
179       TCPFlags = 0;
180       SocketStatus = 0;
181       break;
182     }
183     case SYN_RECD :
184     case ESTABLISHED :
185     {
186       TCPFlags |= TCP_CLOSE_REQUESTED;
187       break;
188     }
189   }
190 }
```

Quellcode 4.19: Bibliothek für den Netzwerkstack (9)

*TCPClose()* aufgerufen werden. Die Verwendung ist unabhängig davon, ob die Verbindung aktiv oder passiv hergestellt wurde. Nach Aufruf der Funktion wird der Zustand der TCP-Session überprüft. Ist ein Aufbau im Gange, so wird dieser unterbrochen und der Zustand der Verbindung auf *CLOSED* gesetzt. Die Flags für Socket und TCP-Session werden zurückgesetzt (Quellcode 4.19, Zeile 176 ff.).

Existiert eine bereits vollständig hergestellte Verbindung, so wird dem Zustandsautomaten mit dem Flag *TCP_CLOSE_REQUESTED* signalisiert, dass die Verbindung bei

der nächsten zyklischen Abarbeitung zu schließen ist (Quellcode 4.19, Zeile 184, Vgl. Abschnitt 4.3.2).

**TCPReleaseRxBuffer()**

Nachdem Nutzdaten empfangen worden sind, muss der Zwischenspeicher vor einer erneuten Übertragung freigegeben werden. Dies verhindert, dass ungewollt Daten durch eingehende Übertragungen überschreiben werden können. Durch den Aufruf der Funktion *TCPReleaseRxBuffer()* wird das Flag *SOCK_DATA_AVAILABLE* des Sockets gelöscht und der Speicher somit freigegeben (Quellcode 4.20).

```
197 void TCPReleaseRxBuffer(void)
198 {
199   SocketStatus &= ~SOCK_DATA_AVAILABLE;
200 }
```

Quellcode 4.20: Bibliothek für den Netzwerkstack (10)

**TCPTransmitTxBuffer()**

Um Daten an Kommunikationsteilnehmer zu senden, muss der Befehl *TCPTransmitTxBuffer()* aufgerufen werden. Befindet sich eine TCP-Session im Zustand *ESTABLISHED* oder *CLOSE_WAIT*, werden die Daten im Zwischenspeicher *TCP_TX_BUF* übertragen (Quellcode 4.21, Zeile 209). Die Länge der Daten, muss zuvor in der Variable *TCPTxDataCount* hinterlegt worden sein. Anschließend wird der Zugriff auf den Buffer gesperrt, indem das entsprechende Flag gesetzt wird (Quellcode 4.21, Zeile 212). Nachdem die Sequenznummern und die Größe des Frames berechnet wurden, wird die Übertragung durch setzen des Flags *SEND_FRAME1* initialisiert (Quellcode 4.21, Zeile 213 ff.). Außerdem wird der Timeout-Timer gestartet (Quellcode 4.21, Zeile 218). Mit der nächsten Ausführung des Zustandsautomaten, werden die Daten übertragen, sofern kein Fehler auftritt.

**ARM CMSIS-DSP**

Die Bibliothek CMSIS-DSP (Cortex Microcontroller Software Interface Standard - Digital Signal Processor ) ist eine von ARM standardisierte Softwarebibliothek. Sie bietet umfangreiche Funktionen für die digitale Signalverarbeitung mit Cortex-M-Mikrocontrollern, welche in vielen Fällen wesentlich weniger Befehlszyklen als die Standardfunktionen von C benötigen. So werden für die Berechnung des Sinus im Float-Format 2085 benötigt, wohingegen die DSP-Bibliothek den Wert nach lediglich 1233 Takten berechnet hat. Darüber hinaus ermöglicht sie eine hohe Portierbarkeit auf Derivate anderer Lizenzfertiger, da diese zu Kompatibilität verpflichtet sind. Die für diese Projekt verwendeten Funktionalitäten sollen hier kurz erläutert werden. Dazu zählt vor allem (Abschnitt 3.4.2) der digitale FIR-Filter zur Erkennung des Synchronimpulses (Abschnitt 4.3.4), welcher

```
207 void TCPTransmitTxBuffer(void)
208 {
209   if ((TCPStateMachine == ESTABLISHED) || (TCPStateMachine == CLOSE_WAIT))
210     if (SocketStatus & SOCK_TX_BUF_RELEASED)
211     {
212       SocketStatus &= ~SOCK_TX_BUF_RELEASED;          // occupy tx-buffer
213       TCPUNASeqNr += TCPTxDataCount;                  // advance UNA
214
215       TxFrame1Size = ETH_HEADER_SIZE + IP_HEADER_SIZE + TCP_HEADER_SIZE +
    TCPTxDataCount;
216       TransmitControl |= SEND_FRAME1;
217
218       LastFrameSent = TCP_DATA_FRAME;
219       TCPStartRetryTimer();
220     }
221 }
```

Quellcode 4.21: Bibliothek für den Netzwerkstack (11)

für verschiedene Zahlenformate implementiert wurde. Darunter auch für das Q-Format (Vgl. Abschnitt 3.4.2).

**FIR-Filter**

Der FIR-Filter besteht grundlegend aus einer Filter-Struktur, welche alle benötigten Informationen und Kenngrößen des Filters enthält. Darüber hinaus existieren Funktionen zur Initialisierung des Filters und zur Berechnung des Filterausgangs. Diese sind in den Zahlenformaten Q7, Q15, Q31 sowie Float implementiert. Da der Filter für diese Anwendung das Q15-Format verwendet, wird nur auf die dafür benötigten Funktionen eingegangen. Die Implementierungen für andere Formate sind in der Funktion jedoch analog.

Die Filterstruktur vom Typ *arm_fir_instance_q15* benötigt die Anzahl der Filtertaps, den Zeiger auf das Array mit den Filterkoeffizienten und einen Zeiger auf ein Array vom Typ *pState*, welches intern zur Berechnung des Ergebnisses benötigt wird. Diese können manuell, oder mit Hilfe der Funktion *arm_fir_init_q15()* erzeugt werden.

Nachdem die Struktur angelegt ist, können die digitalen Informationen gefiltert werden. Dazu wird die Funktion *arm_fir_q15* aufgerufen. Diese verwendet als Paramter die Filterstruktur, einen Zeiger auf die Eingangsdaten, einen Zeiger auf einen Speicherbereich für die auszugebenden Daten und die Anzahl der zu verarbeitenden Samples. Dieses Unterprogramm existiert zusätzlich als „schnelle“ Implementierung mit dem Namen *arm_fir_fast_q15*. Diese erhöht die Performance auf Kosten der Genauigkeit. Kommt es zu einem Überlauf bei mathematischen Operationen, wird das Ergebnis verzerrt, da auf Behandlung eines Überlaufs verzichtet wird.
In Unterabschnitt 4.3.4 ist die Implementierung eines digitalen FIR-Filters im Q15-Format mit 315 Taps zu finden.

```
6  uint8_t BufferInit_uint8_t(buffer_uint8_t *buffer_s,uint32_t size,bool
       init_memory) {
7    (*buffer_s).buffer_size = size;
8    uint8_t *ptr = NULL;
9    if ( init_memory ) {
10     ptr = malloc((*buffer_s).buffer_size*sizeof(unsigned char));
11     if (!ptr) {
12       return BUFFER_FAIL;
13     }
14   }
15   (*buffer_s).read = 0;
16   (*buffer_s).write = 0;
17   (*buffer_s).data = ptr;
18   (*buffer_s).data_size = 0;
19   return BUFFER_SUCCESS;
20 }
```

Quellcode 4.22: Initialisierung des zirkulären Ringspeichers

### Zirkulärer Ringspeicher

Der zirkulär organisierte Ringspeicher benötigt für den Zugriff auf die ihm zugewiesenen Speicherbereiche eine weitere logische Abstraktionsebene, welche durch diese Bibliothek zur Verfügung gestellt wird. Weitere Informationen zum allgemeinen Prinzip des Ringspeichers und zu seiner Funktionsweise sind in Abschnitt 4.3.4 zu finden. Damit physischer Speicher als zirkulärer Ringspeicher verwendet werden kann, müssen der Speicherbereich und einige Metainformationen vorbereitet werden. Die Funktion *BufferInit_uint8_t* reserviert dafür die entsprechende Speichergröße (Quellcode 4.22, Zeile 9 ff.) und legt Zähler für die Schreib- und Lesezugriffe (Quellcode 4.22, Zeile 15 f.), sowie eine Variable, welche die Größe des aktuell belegten Speichers enthält, an (Quellcode 4.22, Zeile 18).

Schreibend kann über das Unterprogramm *BufferIn_uint8_t()* auf den Speicher zugegriffen werden. Dazu werden die Metadaten der Struktur überprüft. Sollte kein freier Speicher mehr zur Verfügung stehen, wird der Vorgang mit einem Fehler abgebrochen (Quellcode 4.23, Zeile 36 ff.). Andernfalls werden die Daten geschrieben und der Zähler wird erhöht. Wird dabei das Ende des physischen Speichers erreicht, wird der Zähler auf dessen Anfang gesetzt. Abschließend wird die Anzahl des belegten Speichers erhöht (Quellcode 4.23, Zeile 44).

Um Informationen aus dem Speicher lesen zu können, muss der Befehl *BufferOut_uint8_t()* ausgeführt werden. Sollten keine Informationen gespeichert und der Speicher somit leer sein, so wird ein Fehler ausgegeben (Quellcode 4.24, Zeile 52 f.). Sind Daten vorhanden, so werden diese in den, als Parameter übergebenen, Zeiger gespeichert. Anschließend werden die Metadaten wie aktuelle Leseposition und die belegte Speichergröße aktualisiert (Quellcode 4.24, Zeile 58 ff.).

```
31 uint8_t BufferIn_uint8_t(buffer_uint8_t *buffer_s,uint8_t byte)
32 {
33
34   if ( ( (*buffer_s).write + 1 == (*buffer_s).read ) ||
35        ( (*buffer_s).read == 0 && (*buffer_s).write + 1 == (*buffer_s).
     buffer_size ) )
36     return BUFFER_FAIL; // voll
37
38   (*buffer_s).data[(*buffer_s).write] = byte;
39
40  (*buffer_s).write++;
41   if ((*buffer_s).write >= (*buffer_s).buffer_size)
42     (*buffer_s).write = 0;
43
44   (*buffer_s).data_size++;
45
46   return BUFFER_SUCCESS;
47 }
```

Quellcode 4.23: Schreiben in den zirkulären Ringspeicher

```
49 uint8_t BufferOut_uint8_t(buffer_uint8_t *buffer_s, uint8_t *pByte)
50 {
51   if ((*buffer_s).read == (*buffer_s).write)
52     return BUFFER_FAIL;
53
54   if (pByte != NULL) {
55     *pByte = (*buffer_s).data[(*buffer_s).read];
56   }
57
58   (*buffer_s).read++;
59   if ((*buffer_s).read >= (*buffer_s).buffer_size)
60     (*buffer_s).read = 0;
61
62   (*buffer_s).data_size--;
63
64   return BUFFER_SUCCESS;
65 }
```

Quellcode 4.24: Lesen aus dem zirkulären Ringspeicher

Die hier aufgeführten Funktionen dienen der Verwaltung von Speicher mit 8 Bit großen Daten. Analog wurden in die Bibliothek Unterprogramme für das Management von Ringspeichern für den Datentyp *short int*, *float*, sowie *Q7* und *Q15* aufgenommen.

### 4.3.3 Sender

Die bisher eröterten Lösungen wurden in der folgend beschriebenen Firmware umgesetzt. Neben der Digitalisierung des Audiosignals, erfolgt sowohl der Versand des für die Synchronisation benötigten Impulses, als auch die Ausgabe der digitalisierten Informationen.

#### Initialisierung

Die Initialisierung ist nötig um Softwarevariablen, sowie die Hardware und die Peripherie des Controllers für den Betrieb zu konfigurieren und anschließend erstmalig in Betrieb zu nehmen. Dabei kommen Präprozessoranweisungen, globale, sowie lokale Variablen und Funktionen zum Einsatz. Quellcode 4.25 zeigt die Präprozessoranweisungen der Firmware des Senders.

```
1  #include <main.h>
2
3  #define SYNC_SIZE 100
4  #define SAMPLING_RATE 16000
5  #define SYNC_FREQ 1600
6  #define BUFFERSIZE (SAMPLING_RATE-SYNC_SIZE)
7  #define KEYSIZE 40  // default keysize
8
9  #if KEYSIZE > 0
10 #define BLOCKSIZE ((BUFFERSIZE+SYNC_SIZE)/KEYSIZE)
11 #else
12 #define BLOCKSIZE 1
13 #endif
```

Quellcode 4.25: Präprozessoranweisungen der Initialisierungsphase

Zu Beginn werden über die header-Datei *main.h* alle benötigten Bibliotheken geladen. Darin enthalten sind einige Standardbibliotheken von C, sowie proprietäre Bibliotheken von ARM und KEIL und eigene, für diese Hardware erstellte, Bibliotheken. Die Bibliotheken wurden in Unterabschnitt 4.3.2 bereits ausführlich erläutert.

Anschließend folgt die Definition einiger Symbole, welche bei Benutzung im Quelltext vor der Übersetzung vom Compiler ersetzt werden. Sie dienen der grundlegenden Konfiguration der Software und ermöglichen eine hohe Flexibilität, ohne umfangreiche Änderungen im Quellcode vornehmen zu müssen. Tabelle 4.3 gibt eine Übersicht über die verwendeten Symbole und ihre Bedeutung für den Ablauf des Programms.

| Symbol | Bedeutung |
|---|---|
| SYNC_SIZE | Länge des Synchronimpulses in Samples |
| SAMPLING_RATE | Abtastrate in Herz |
| SYNC_FREQ | Frequenz des Synchronimpulses |
| BUFFERSIZE | Länge des Datenspeichers, berechnet sich aus SAMPLING_RATE und SYNC_SIZE |
| KEYSIZE | Länge des verwendeten Kryptoschlüssels |
| BLOCKSIZE | Länge eines Verschlüsselungssymbols, berechnet sich aus BUFFERSIZE, SYNC_SIZE und KEYSIZE |

Tabelle 4.3: Symbolische Konstanten und ihre Bedeutung

Neben den symbolischen Konstanten werden einige globale Variablen benötigt um die funktionsübergreifende Kommunikation zu gewährleisten. Dazu zählen Buffer- und Zustandsvariablen für das digitalisierte Informationssignal, den Netzwerk-Stack, den Automaten des Hauptprogramms, sowie den Snychronisationsalgorithmmus und den Taster zur Auswahl des Automatenzustandes (Quellcode 4.26). Tabelle 4.4 gibt eine Übersicht über die globalen Variablen.

```
14 bool volatile send_sync = false;
15 int volatile dac_val = 0;
16
17 unsigned char *key;
18 unsigned short data_counter = 0;
19
20 unsigned char enc_buffer[BUFFERSIZE];
21 unsigned char dac_buffer[BUFFERSIZE];
22
23 bool STATE = 1;
24 bool button_prev_val = 1;
25
26 int *sinus;
27
28 int keysize = KEYSIZE;
29
30 bool send_data = false;
31 unsigned char rx_string[256];
32 unsigned char tx_string[256];
```

Quellcode 4.26: Globale Definition von Variablen in der Initialisierungsphase

Die Variable *send_sync* signalisiert, dass der Eingangsbuffer gefüllt und somit die Zeit zwischen zwei Synchronimpulsen verstrichen ist. Erhält diese boolesche Variable den Wert *true*, wird der Synchronimpuls übertragen. Während dieses Zeitraums werden kei-

| **globale Variable** | **Typ** |
|---|---|
| send_sync | Zustandsvariable |
| SAMPLING_RATE | Abtastrate in Frequenz |
| SYNC_FREQ | Frequenz des Synchronimpulses |
| BUFFERSIZE | Länge des Datenspeichers, berechnet sich aus SAMPLING_RATE und SYNC_SIZE |
| KEYSIZE | Länge des verwendeten Kryptoschlüssels |
| BLOCKSIZE | Länge eines Verschlüsselungsblocks, berechnet sich aus BUFFERSIZE, SYNC_SIZE und KEYSIZE |

Tabelle 4.4: Globale Variablen und deren Verwendung

ne Daten am Analogeingang des Controllers eingelesen. Er ist folglich so kurz wie möglich zu halten, um eine akustische Wahrnehmung zu verhindern. Der Grenzwert dafür liegt nach Abschnitt 2.6 bei $20\text{ms}$. Mit der gewählten Länge von 100 Samples und der Abtastfrequenz von $16\text{kHz}$ ergibt sich jedoch lediglich eine Dauer von $6.25\text{ms}$. Die durch die Synchronisation bedingte Unterbrechung ist für den durchschnittlichen Hörer demnach nicht wahrnehmbar.

Zwei weitere Variablen dienen der Digitalisierung und Ausgabe des Audiosignals. *dac_val* beinhaltet den jeweiligen, über den DAC, auszugebenen Digitalwert, wohingegen mit *data_counter* die bereits ausgegeben Samples gezählt werden um zum einen die Zustandsvariable für das Senden des Synchronimpulses setzen zu können und zum anderen einen Überlauf der Nutzdatenspeicher zu verhindern.

Für die Steuerung des Zustandsautomaten werden die Variablen *STATE* und *button_prev_val* verwendet. Wird der Tasterzustand abgefragt, so wird der aktuelle Zustand in *button_prev_val* gespeichert, um ihm bei einer erneuten Abfrage vergleichen und eine eventuelle Änderung detektieren zu können. In *STATE* wird schließlich der Zustand des Automaten gespeichert, falls eine Tasterbetätigung erkannt wurde (Vgl. Unterabschnitt 4.3.1).

Den Pointern *key* und *sinus* werden im Laufe der Initialisierung die jeweiligen Adressen für die Speicherbereiche zugewiesen, in denen der Kryptographieschlüssel und die Wertetabelle für das sinusförmige Snychronisationssignal hinterlegt sind. Diese werden dynamisch während der Initialisierung erzeugt. Um einen funktionsübergreifenden Zugriff zu gewährleisten, werden diese beiden Pointer global angelegt.
Die eingelesenen Daten werden im Array *enc_buffer* verschlüsselt abgelegt, sofern die Schlüssellänge größer als null ist. Zur Ausgabe werden sie in den Buffer *dac_buffer* kopiert.

In den Arrays *rx_string* und *tx_string* werden die über das Netzwerk zu empfangenden und zu sendenden Daten gespeichert beziehungsweise vorgehalten. Über die Zustandsvariable *send_data* wird der Versand der Daten eingeleitet.

Im Anschluss beginnt die Abarbeitung des Hauptprogramms mit der tatsächlichen Initialisierung des Systems. Zu Beginn werden die in der Software hinterlegten Kryptographieschlüssel definiert und entsprechend der Auswahl über die Präprozessoranweisungen in den Speicherbereich des aktuellen Schlüssels geladen (Quellcode 4.27, Zeile 35 - 58). Darüber hinaus wird über die Funktion *sinus_init()* die Wertetabelle für das Synchronisationssignal erzeugt und im Speicher hinterlegt (Quellcode 4.27, Zeile 60). Da das Sinussignal periodisch aufgerufen wird und somit eine zyklische Berechnung der Werte nötig wäre, wurde auf eine feste Wertetabelle zurückgegriffen um die Laufzeit des Programms zu reduzieren. Das Laden der Werte aus dem Speicher benötigt wesentlich weniger Takte als die Berechnung mit der Sinus-Funktion der C-Bibliothek. Der Aufruf der Funktion erfolgt mit der gewünschten Auflösung einer Periode als Parameter. Innerhalb der Funktion wird anschließend der, entsprechend der Auflösung benötigte, Speicher reserviert. Nachdem die Wertetabelle abgelegt wurde, wird der Zeiger, welcher auf diesen Speicherbereich zeigt, zurückgegeben (Quellcode 4.28).

Damit die Kommunikation des Mikrocontrollers mit dem A/D-Wandler gelingt, muss der SSP (Synchronous Serial Port) zuvor konfiguriert werden (Quellcode 4.27, Zeile 62). Dazu werden die Funktionen der entsprechenden Bibliothek genutzt (Vgl. Abschnitt 4.3.2).

Nach der erfolgreichen Konfiguration des seriellen Interfaces erfolgt die Initialisierung des D/A-Wandlers (Quellcode 4.27, Zeile 63). Die dafür angelegte Bibliothek stellt die benötigte Funktion (*init_dac()*) zu Verfügung (Vgl. Abschnitt 4.3.2).

Bevor der Nutzer Feedback durch das verbaute LC-Display erhalten kann, müssen die Controller entsprechend konfiguriert werden. Nach der Initialisierung durch das Unterprogramm *lcd_init()* wird der Inhalt des Displays durch den Befehl *lcd_clear()* gelöscht (Quellcode 4.27, Zeile 65 f.). Anschließend können erste Informationen an den Nutzer ausgegeben werden. Hierzu wird zuerst der TCP-Port für die Kommunikation im Konfigurationsmodus festgelegt (Quellcode 4.27, Zeile 68) und anschließend mit einer Meldung, die auf die Initialisierung des Netzwerkstacks verweist, auf dem Display ausgegeben (Quellcode 4.27, Zeile 70 ff.). Mit der Funktion *lcd_cursor()* wird die Position der darzustellenden Zeichen festgelegt, anschließend werden mit Hilfe von *lcd_putstring()* die gewünschte Zeichenkette ausgegeben.

Mit dem Erlöschen der Meldung, das den Abschluss der Netzwerkinitialisierung signalisiert, wird der Portpin P1.31 als Eingang definiert (Quellcode 4.27, Zeile 83) und der Timer des Mikrocontrollers als Basis für die Datenverarbeitung gestartet. Der Portpin dient als Eingang für den Taster, über den die Zustandswahl des Automaten ermög-

```
35 int main(void) {
36
37   unsigned char key10[10] = {9, 7, 5, 3, 1, 8, 6, 4, 2, 0};
38   unsigned char key20[20] = {1 ,10 ,7 ,13 ,5 ,8 ,11 ,14 ,17 , 9, 0, 3, 2, 4,
       12 ,19 ,15 ,16 ,18 ,6 };
39   unsigned char key40[40] = {23 ,6 ,39 ,11 ,33 ,15 ,5 ,3 ,8 ,30 ,25 ,38 ,18
       ,19 ,35 ,0 ,37 ,1 ,4 ,20 ,27
       ,9,32,29,12,7,24,28,14,34,21,10,2,36,26,22,17,13,16,31};
40
41   if (keysize) {
42     key = (unsigned char*)malloc(40);
43   }
44
45   // init key for encryption
46   switch(keysize) {
47     case 10:
48       memcpy(key,key10,10);
49       break;
50     case 20:
51       memcpy(key,key20,20);
52       break;
53     case 40:
54       memcpy(key,key40,40);
55       break;
56     default:
57       break;
58     }
59
60   sinus = sinus_init(100);
61
62   spi_init();
63   init_dac();
64
65   lcd_init();
66   lcd_clear();
67
68   TCPLocalPort = 80;          // TCP/IP-Port
69
70   char string[40];
71
72   sprintf(string,"Starting networking on port %d ...",TCPLocalPort);
73
74   lcd_cursor(1,1);
75   lcd_putstring((unsigned char*)string);
76
77   free(string);
78
79   TCPLowLevelInit();          // TCP/IP-Init
80
81   lcd_clear();
82
83   LPC_GPIO0->FIODIR &= ~(1UL<<6); // Set P20/P1.31 to Input
84
85   timer1_init();              //init timer1 with 16Khz - base for 1s windows
86
87   char mystring[2] = "0";
88
89   int button_val = 0;
```

Quellcode 4.27: Initialisierungsanweisungen im Hauptprogramm des Senders

```
#include "LPC17xx.h"                        // Device header
#include <sinus.h>


int* sinus_init(int resolution) {
  int* sinus_buffer=malloc(resolution*sizeof(int));
  for(int i=0;i<resolution;i++) {
    *(sinus_buffer+i)=(512*((0.4*sin((2*3.14159265359/(resolution-1)*i)))))
    +512;
  }
  return sinus_buffer;
}
```

Quellcode 4.28: Bibliothek zur Erzeugung der Wertetabelle eines Sinussignals

licht wird. Sobald eine Null, welche den aktuellen Zustand des Tasters anzeigt, auf dem Display erscheint, ist die Initialisierung des Systems beendet und der Betriebsmodus erreicht.

### Synchronisation

Kernstück der Entwicklung stellt die Synchronisation der beiden Kommunikationspartner dar. Nachdem in Abschnitt 3.3 mögliche Verfahren und in Unterabschnitt 3.4.2 die präferierte Lösung vorgestellt worden sind, soll nun auf die Implementierung in der Firmware eingegangen werden. Um die Synchronisation der Teilnehmer zu ermöglichen, fügt der Sender einen Sinus-Impuls in die Nutzdaten ein (Abbildung 4.18). Dazu wird der Nutzdatenstrom unterbrochen. Dies geschieht durch die ISR, des als Zeitbasis gestarteten, Timers.

Die ISR wird entsprechend der Abtastrate für jedes zu verarbeitende Sample aufgerufen. Die Aufrufe werden über die Variable *data_counter* gezählt. Erreicht der Wert die Größe des Datenspeichers (*BUFFERSIZE*), so ist der Zeitrahmen von einer Sekunde für die Synchronisation erreicht und das entsprechende Flag (*send_sync*) wird gesetzt (Quellcode 4.29, Zeile 170 ff.). Somit findet beim nächsten Aufruf der ISR die Synchronisation statt.
Während der Synchronisation blockiert die Ausführung des Codes das Einlesen weiterer Daten. Dies ist dadurch bedingt, dass dies in der gleichen ISR geschieht. Ist das entsprechende Flag gesetzt, so werden insgesamt zehn Perioden des sinusförmigen Synchronimpulses mit einer Frequenz von $1,6\mathrm{kHz}$ ausgegeben. Dazu werden die entsprechenden Funktionswerte der Winkelfunktion aus der Wertetabelle abgerufen (Quellcode 4.29, Zeile 151 ff.). Diese wurde während der Initialisierung des Systems angelegt (Vgl. Abschnitt 4.3.3). Die Frequenz wird über die Verzögerung eingestellt, die zwischen der Ausgabe zweier Werte über den DAC liegt (Quellcode 4.29, Zeile 156 ff.). Die Bestimmung dieser Verzögerung erfolgte empirisch über die Untersuchung des Signals mit einem Oszilloskop.

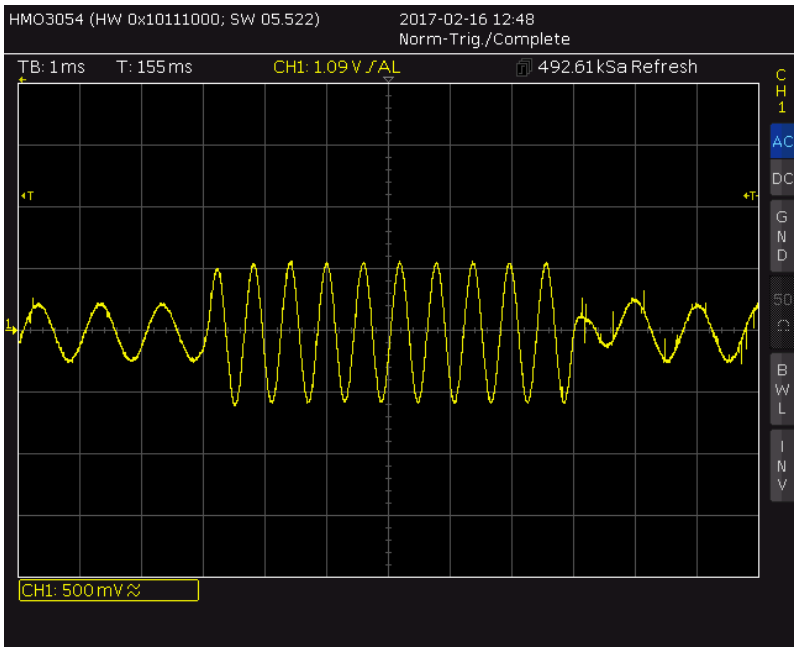



Abbildung 4.18: In Nutzsignal eingefügter Synchronimpuls

Anschließend an die Ausgabe des Synchronimpulses werden die Daten des Eingangszwischenspeichers in den Ausgangszwischenspeicher kopiert. Nachdem das Flag und der Zähler zurückgesetzt worden, beginnt nun das Einlesen der Daten für einen Zyklus von vorn.

### Verschlüsselung

Die Verschlüsselung selbst ist kein Gegenstand der Arbeit, wird jedoch benötigt um eine qualitative Aussage über das Synchronisationsverfahren treffen zu können. Eine Verschlüsselung der Daten muss stattfinden um die Genauigkeit des Verfahrens bestimmen zu können und um einen subjektiven Höreindruck zu bekommen. Aus diesem Grund wurde das Verfahren der Transposition, auch Scrambling genannt, implementiert (Vgl. Abschnitt 2.7.1). Neben der Klartextübertragung bietet die Firmware die Möglichkeit drei vorgegebene Schlüssel verschiedener Länge zu verwenden. Wahlweise können über den Konfigurationsmodus neue Schlüssel mit gleicher Länge übertragen und für die aktuelle Betriebssitzung gespeichert werden.
Die Verschlüsselung der Daten erfolgt direkt nach der Digitalisierung des Analogsignals. Dazu muss das Flag zum Senden des Synchronsignals gelöscht, und ein Kryptographieschlüssel mit einer Länge größer null ausgewählt sein (Quellcode 4.29, Zeile 139 f.). Ist dies der Fall werden die eingelesenen Daten direkt an die, durch die Transposition festgelegte, Position gespeichert. Diese wird aus der aktuellen Position im Speicher, der Blockgröße und dem Kryptographieschlüssel berechnet. Dadurch kann die Ausgabe Verzögerungsfrei erfolgen, sobald der Buffer gefüllt wurde (Quellcode 4.29, Zeile 141). Sollen komplexere Blockchiffren implementiert werden, können die eingelesenen

```
134 void TIMER1_IRQHandler(void) {
135   NVIC_DisableIRQ(TIMER1_IRQn);
136
137   LPC_TIM1->MR0 = 1329;                        // 16000 Hz = SampleRate
138
139   if (!send_sync) {
140     if (keysize>0) {
141       enc_buffer[(key[data_counter/BLOCKSIZE]*BLOCKSIZE)+data_counter%(
      BLOCKSIZE)] = (RecSPI()>>2);
142     }
143     else {
144       enc_buffer[data_counter] = (RecSPI()>>2);
145     }
146
147     LPC_DAC->DACR = (dac_buffer[data_counter]<<8);
148
149     data_counter++;
150   }
151   else {
152     for(int i = 0; i < 10; i++) {
153       for(int z = 0; z < 100; z++) {
154         dac_val = (*(sinus+z));
155         LPC_DAC->DACR = (dac_val<<6);
156         for(int y =0; y<141;y++);
157         __nop();
158         __nop();
159       }
160     }
161
162     //copy old data to output buffer for dac
163     memset(dac_buffer,116,BUFFERSIZE);
164     memcpy(dac_buffer,enc_buffer,BUFFERSIZE);
165     memset(enc_buffer,116,BUFFERSIZE);
166     data_counter = 0;
167     send_sync = false;
168   }
169
170   if (data_counter == BUFFERSIZE-1) {           // after 1s - t(sync_signal)
171     data_counter = 0;
172     send_sync = true;
173   }
174
175   LPC_TIM1->TCR |= (1<<1);
176   LPC_TIM1->IR  |= (1<<0);                      // Clear MR0 interrupt flag
177   LPC_TIM1->TCR  = (1<<0);
178   NVIC_EnableIRQ(TIMER1_IRQn);
179 }
```

Quellcode 4.29: Interrupt Service Routine des Timers

Daten zuvor in einem weiteren Zwischenspeicher gehalten werden, bis die gewünschte Blockgröße erreicht ist. Die Architektur ermöglicht eine einfache Implementierung, welche keine großen Änderungen am Quellcode erfordern.
Soll die Übertragung im Klartext erfolgen, so werden die Daten in unveränderter Reihenfolge in den Speicher geschrieben (Quellcode 4.29, Zeile 143 f.).
Abschließend erfolgt die Ausgabe der Daten des entsprechenden Zwischenspeichers über den DAC (Quellcode 4.29, Zeile 147).

#### Betriebsmodus

Der Betriebsmodus ist der Zustand des Automaten, in den direkt nach der Initialisierung übergegangen wird. Ist dieser aktiv, so werden die Daten verschlüsselt, falls ein Kryptoschlüssel ausgewählt ist und über den DAC ausgegeben. Dazu wird der Interrupt für den Timer aktiviert, sobald der Modus gewählt wird. Die Umschaltung zwischen Betriebs- und Konfigurationsmodus erfolgt über einen Taster, dessen Zustand bei jeder Abarbeitung des Hauptprogramms geprüft wird. Beim Übergang in den Konfigurationsmodus wird der Interrupt deaktiviert und die Datenverarbeitung damit gestoppt. (Quellcode 4.30, Zeile 93 und 108 ff.).

#### Konfigurationsmodus

Wurde die Variable *STATE* durch betätigen des Tasters geändert, so wechselt der Automat in den Konfigurationsmodus, indem der Interrupt des Timers deaktiviert wird (Quellcode 4.30, Zeile 103 ff.). Anschließend werden eventuell offene Sockets geschlossen um einen Speicherzugriffsfehler zu vermeiden. Nachdem alle Sockets frei sind, kann ein neuer Listener gestartet werden (Quellcode 4.31, Zeile 124 f.). Anschließend wird der implementierte Serverdienst ausgeführt. Dieser dient der Übertragung der Kryptoschlüssel von der Anwendungssoftware auf dem PC an die Firmware.

#### Netzwerkserver

Der Serverdienst lauscht auf einem Socket, verarbeitet empfangene Daten sofern sie dem Protokoll entsprechen und bestätigt diese. Dazu wird überprüft, ob Daten zur Verarbeitung vorliegen und kein Versand ansteht. Anschließend werden diese in entsprechende Zwischenspeicher kopiert (Quellcode 4.32, Zeile 193 ff.).

Das Protokoll unterscheidet zwischen zwei unterschiedlichen Datenpaketen. Zum einen besteht die Möglichkeit Kryptoschlüssel an den Server zu übertragen. Dies geschieht durch die Übertragung eines Frames, der dem Format von Tabelle 4.5 entspricht. Wurde dieser Frame erkannt, werden die Schlüssellänge und der empfangene Kryptoschlüssel extrahiert und im Speicher hinterlegt (Quellcode 4.32, Zeile 198 ff.). Die Länge des

```
91  while(true) {
92
93    button_val = ((LPC_GPIO0->FIOPIN)>>6)&0x01;
94
95    sprintf(mystring,"%i",STATE);
96
97    lcd_cursor(1,1);
98    lcd_putstring((unsigned char*)mystring);
99
100   if (button_prev_val != button_val) {
101     if (!button_val) {
102       STATE = !STATE;    //if button was pressed change mode, else ignore
103       if (STATE) {
104         NVIC_EnableIRQ(TIMER1_IRQn);
105       }
106       else {
107         NVIC_DisableIRQ(TIMER1_IRQn);
108       }
109
110       TCPClose();
111     }
112     button_prev_val = button_val;                    // save button level
113     for(int i=0;i<10000;i++);                        // debounce
114   }
115
116   if (STATE) {
117       lcd_cursor(1,2);
118       lcd_putstring((unsigned char*)"call mode  ");
119   }
```

Quellcode 4.30: Hauptschleife der Firmware mit Betriebsmodus

```
120   else {
121     lcd_cursor(1,2);
122     lcd_putstring((unsigned char*)"config mode");
123
124     if (!(SocketStatus & SOCK_ACTIVE)) {
125       TCPPassiveOpen();    // listen for incoming TCP-connection //
126     }
127     Server();
128     DoNetworkStuff();      // handle network and easyWEB-stack
129   }
130 }
```

Quellcode 4.31: Hauptschleife der Firmware, Konfigurationsmodus

| **Byte** | **0** | **1** | **2** | **...** | **41** |
|---|---|---|---|---|---|
| **Inhalt** | s | Schlüssellänge | Schlüssel | | |
| **Länge** | 1 | 1 | 40 | | |

Tabelle 4.5: Protokoll zum Schreiben der Konfiguration

Schlüssels ist dabei bytecodiert, sodass Längen bis 255 Zeichen in einem Byte übertragen werden können. Das System verwendet diesen nun für die Verschlüsselung. Anschließend werden die Daten aus dem Speicher gelesen und an den Benutzer zurückgesendet (Quellcode 4.32, Zeile 204 ff.). Dies ermöglicht die optische Verifikation der Übertragung durch diesen, da die Daten in der Anwendungssoftware visualisiert werden. Stimmen sie mit den versendeten überein, war die Konfiguration erfolgreich. Um den Datenversand einzuleiten, wird das Flag *send_data* gesetzt (Quellcode 4.32, Zeile 208).

```
193     if(SocketStatus & SOCK_DATA_AVAILABLE && !send_data)   // daten im
        Enpfangsbuffer vorhanden
194     {
195       memcpy(rx_string,TCP_RX_BUF,TCPRxDataCount);
196       memcpy(tx_string,TCP_RX_BUF,TCPRxDataCount);
197
198       if (rx_string[0] == 's' && TCPRxDataCount == 42) {
199         // set-string
200         //free(key);
201         keysize = rx_string[1];
202         //key = malloc(keysize*sizeof(char));
203         memcpy(key,rx_string+2,keysize);
204         tx_string[0] = 's';
205         tx_string[1] = keysize;
206         memcpy(tx_string+2,key,keysize);
207
208         send_data = true;
209       }
```

Quellcode 4.32: Konfigurationsmodus, Schreiben von Kryptoschlüsseln

Zum anderen können die aktuellen Einstellungen vom Benutzer abgerufen werden. Dazu wird das entsprechende Paket an den Server gesendet, welcher anschließend die Daten überträgt. Entspricht der Inhalt des empfangenen Frames dem Protokoll (Tabelle 4.6), werden Schlüssellänge und der Schlüssel selbst in den Sendespeicher des TCP/IP-Stacks kopiert und das Flag zum Senden gesetzt (Quellcode 4.33, Zeile 210 ff.). In beiden Fällen wird der Emfpangsbuffer am Ende freigegeben.
Im nächsten Schritt wird die Funktion *DoNetworkStuff()* aufgerufen, welche die Daten über die Ethernetschnitstelle überträgt (Vgl. Abschnitt 4.3.2).

| Byte | 0 | 1 | 2 | ... | 41 |
|---|---|---|---|---|---|
| Inhalt | r | Schlüssellänge | Schlüssel | | |
| Länge | 1 | 1 | 40 | | |

Tabelle 4.6: Protokoll zum Lesen der Konfiguration

```
210       else if ((!strcmp((char*)&rx_string[0],"r")) && TCPRxDataCount == 42){
211         // read-string
212         tx_string[0] = 'r';
213         tx_string[1] = keysize;
214         memcpy(tx_string+2,key,keysize);
215
216         send_data = true;
217       }
218       TCPReleaseRxBuffer();
```

Quellcode 4.33: Konfigurationsmodus, Lesen von Kryptoschlüsseln

### 4.3.4 Empfänger

Die Firmware des Empfängers unterscheidet sich in einigen wesentlichen Punkten von der des Senders. Diese beziehen sich vor allem auf den Vorgang der Synchronisation und dadurch auf die Initialisierung der Firmware. Der Empfänger hat die Aufgabe den vom Sender ausgesendeten Synchronisationsimpuls im Signal zu detektieren und die Verschlüsselung darauf zu Synchronisieren. Dies ist nötig um die Informationen korrekt entschlüsseln zu können.

#### Initialisierung

Die Initialisierung gleicht grundlegend der des Senders (Vgl. Abschnitt 4.3.3), wesentliche Unterschiede ergeben sich jedoch durch die Initialisierung des digitalen Filters zur Erkennung des Synchronimpulses, sowie der Initialisierung des Verschlüsselungsverfahrens.
Da die Abarbeitung des Filters viel Rechenzeit in Anspruch nimmt, müssen möglichst viele Berechnungsschritte außerhalb der Hauptschleife der Firmware vorgenommen werden. So werden z.b. die Klartextpositionen der transponierten Symbole berechnet und in einem Array gespeichert. Bei der späteren Entschlüsselung müssen diese lediglich aus dem Array abgerufen werden.

Auch die Firmware des Empfängers enthält einige Präprozessoranweisungen zur Konfiguration des Systems. Die Symbole *BUFFERSIZE*, *SYNC_SIZE* und *KEYSIZE* haben die gleiche Bedeutung wie beim Sender. *FILTERSIZE* enthält die Anzahl der Filtertaps, die für die Erkennung des Synchronimpulses verwendet werden (Quellcode 4.34, Zeile 3 ff.). Abschließend erfolgt eine Direktive zur Berechnung der Symbolgröße. Dabei wird

aus Speicher- und Schlüssellänge die Länge einzelner Symbole in Samples berechnet (Quellcode 4.34, Zeile 9 ff.). Diese wird später Bei der Entschlüsselung der Blöcke benötigt.

```
1  #include <main.h>
2
3  #define BUFFERSIZE 16000
4  #define SYNC_SIZE 100
5  #define FILTERSIZE 315
6  #define KEYSIZE 40 // default keysize
7
8
9  #if KEYSIZE > 0
10 #define BLOCKSIZE (BUFFERSIZE/KEYSIZE)
11 #else
12 #define BLOCKSIZE 1
13 #endif
```

Quellcode 4.34: Präprozessoranweisungen der Firmware des Empfängers

Die Filterfunktion der Bibliothek CMSIS-DSP erlaubt die Anzahl der pro Aufruf zu verarbeitenden Samples festzulegen. Da zu jedem Sample das aktuelle Ergebnis am Ausgang des Filters bereit stehen soll, beträgt diese 1 (Quellcode 4.35, Zeile 60).

Darüber hinaus werden einige Buffer benötigt, so jeweils einer für die entschlüsselten Daten am Eingang des Systems, sowie einer für die Ausgabe der Daten über den DAC. Ein weiterer dient der Mittelung der Länge des Synchronimpulses um transiente Störungen und Fehler bei der Erkennung abschwächen zu können. Außerdem verwendet die Firmware ein Schieberegister zur Verzögerung der Werte am Eingang. Dies ist nötig um eine parallele Verarbeitung zwischen dem Filterausgang und dem Nutzsignal zu erreichen.
Wesentlich für die Synchronisation ist der Zwischenspeicher *OutputBufferLast*, mit dessen Hilfe der Mittelwert des Ausgangssignals gebildet und der Zeitpunkt der Synchronisation bestimmt wird. Alle Speicher sind als zirkuläre Ringbuffer organisiert um die Geschwindigkeit bei Schreib- und Leseoperationen zu optimieren (Quellcode 4.35, Zeile 65 ff.).

Anschließend wird die Struktur des digitalen Filters instanziert. Dazu wird ein Statusarray definiert, welches unter anderem in der für die Berechnung des Filters benötigten Instanz Verwendung findet (Quellcode 4.35, Zeile 74 ff.).
Die Mittelwerte der zuvor definierten Zwischenspeicher werden in den Variablen *filter_output_mean* und *mean_prev* festgehalten (Quellcode 4.35, Zeile 77 ff.).
Für die Synchronisation und deren Management werden einige weitere Variablen benötigt. Dazu zählen unter anderem die theoretische Länge des Synchronimpulses und die Dauer während der eine erneute Erkennung verhindert wird, falls ein Impuls bereits erkannt wurde (Quellcode 4.35, Zeile 81 ff.).

```
60 const int block_size = 1;   // we need filtered sample with every clock
61 const int filter_length = FILTERSIZE;
62
63 unsigned int volatile ADCInputVal = 0;
64 unsigned short volatile DACOutputVal = 0;
65 buffer_uint16_t sync_counter_buffer;     //keeps the measured synclength
66 uint8_t dacoutbuffer_b[BUFFERSIZE] __attribute__((at(LPC_RAM_BASE+256)));
       // buffer for outgoing data
67 uint8_t decryptbuffer[BUFFERSIZE] __attribute__((at(LPC_AHBRAM0_BASE)));
68
69 buffer_uint8_t dacoutbuff;
70 buffer_uint8_t delay_buffer;
71 buffer_q15_t InputMeanBuffer;
72 buffer_q15_t OutputBufferLast;
73
74 q15_t fir_state[filter_length + block_size];  //q7-structure for fir-filter
75 arm_fir_instance_q15 FIR_BP = {filter_length, fir_state, (q15_t*)
       filter_coefficients};     // create instance fir struct
76
77 q31_t output_power = 0;
78 q15_t filter_output_mean = 0;
79 q7_t mean_prev = 0;
80 int sync_mean = 0;
81
82 unsigned short int sync_valid = 300;
83 unsigned short int decrypt_counter = 0;
84 unsigned char current_word = 0;
85 unsigned short int sync_length = 179;
86 unsigned short int sync_start = BUFFERSIZE-180;
87 unsigned char* key;
```

Quellcode 4.35: Globale Variablen der Firmware des Empfängers (1)

Das Array *current_words* enthält die Klartextpositionen der Entsprechenden Positionen des Keys, es stellt sozusagen dessen Invertierung dar. Die Variablen *sync_counter*, *sync_high* und *sync* diesen der Kontrolle der Synchronisation, darin wird festgehalten ob der aktuelle Impuls präsent ist, wie lange er dies ist und ob es ausreichend lange war, um ihn als solchen zu erkennen (Quellcode 4.36).

Anschließend beginnt die eigentliche Initialisierungsphase. Abgesehen von der Initialisierung der Buffer und der Berechnung des inversen Kryptoschlüssels, entspricht sie der des Senders.

Wie zuvor angedeutet, muss die Klartextposition der verschlüsselten Daten vor der Entschlüsselung zuvor aus dem Kryptographieschlüssel berechnet werden. Dazu werden die Indizes mit dem Inhalt des entsprechenden Feldes vertauscht und anschließend aufsteigend sortiert. Da bei jedem eingelesenem Sample überprüft werden muss, welche die zugehörige Position im Speicher ist, wird der Schlüssel bereits vor der Hauptschleife berechnet und im Speicher hinterlegt (Quellcode 4.37).

```
90  bool sync = false;
91
92  unsigned char temp = 0;
93
94  arm_status init_result = -6;
95
96  int sync_counter = 0;
97  bool sync_high = false;
98
99  char str1[20];
100
101 #if KEYSIZE>0
102   char current_words[KEYSIZE];
103 #else
104   char current_words[] = {1};
105 #endif
106
107 bool STATE = 1;
108 bool button_prev_val = 0;
109
110 bool send_data = false;
111 unsigned char rx_string[256];
112 unsigned char tx_string[256];
```

Quellcode 4.36: Globale Variablen der Firmware des Empfängers (2)

```
139   // calculte current word if encryption is enabled
140   if (keysize>0) {
141     for (unsigned char z=0;z<keysize;z++) {
142       for(unsigned char i=0;i<keysize;i++) {
143         if (key[i] == z) {
144           current_words[z] = i;
145           break;
146         }
147       }
148     }
149   }
```

Quellcode 4.37: Invertierung des Kryptographieschlüssels

Während der Definition der globalen Variablen wurden bereits die Strukturen für die zirkulären Zwischenspeicher festgelegt. Nun müssen mit Hilfe der entsprechenden Funktionen der Bibliothek noch die dafür benötigen Speicherbereiche angefordert werden (Quellcode 4.38). Einzig der Zwischenspeicher für die Ausgabe der Daten wurde bereits auf einer bestimmten Position im RAM hinterlegt um Konflikte aufgrund der Größe zu vermeiden. Der dazugehörige Zeiger wird an dieser Stelle in der Struktur gespeichert und darüber zugänglich gemacht (Quellcode 4.38, Zeile 190). Sollte es bei der Speicheranforderung zu Fehlern kommen, wird das Programm mit einer entsprechenden Meldung beendet. War diese hingegen erfolgreich, so geht das Programm in die Hauptschleife über und der Automat befindet sich im Zustand des Betriebsmodus.

```
177   if (!BufferInit_q15_t(&InputMeanBuffer,filter_length)) {
178     return EXIT_FAILURE;                  // init buffer struct
179   }
180
181   if (!BufferInit_uint8_t(&delay_buffer,(SYNC_SIZE+(FILTERSIZE<<1))>>1,true)
        ) {
182     return EXIT_FAILURE;
183   }
184
185   if (!BufferInit_uint8_t(&dacoutbuff,BUFFERSIZE,false)) {
186     return EXIT_FAILURE;
187   }
188   else {
189     dacoutbuff.data = dacoutbuffer_b;
190   }
191
192   if (!BufferInit_q15_t(&OutputBufferLast,filter_length)) {
193     return EXIT_FAILURE;
194   }
195
196   if (!BufferInit_uint16_t(&sync_counter_buffer,5)) {
197     return EXIT_FAILURE;
198   }
```

Quellcode 4.38: Initialisierung der zirkulären Ringbuffer

### Prinzip des zirkulären Ringbuffers

Bei zirkulären Ringspeichern handelt es sich um ein Konzept, dass von der üblichen Organisation von Speicher abweicht. Register, oder Speicherbereiche werden häufig wie gewöhnliche Schieberegister behandelt. Dabei wird der Speicher in der Regel nach dem „fifo“-Prinzip (first in first out) gefüllt. Dass heißt, dass dieser beginnend bei der letzten freien Adresse belegt wird. Ist der Speicher voll, so wird das erste gespeicherte Element, das sich auf der letzten Adresse befindet, ausgegeben und der restliche Inhalt entsprechend eine Position nach hinten verschoben. So entsteht am Anfang ein freier Platz für neue Daten. Dieses Pradigma erfordert allerdings eine hohe Anzahl an Speicherzugriffen, da alle Daten einmal gelesen und an eine neue Position im Speicher geschrieben werden müssen, sobald dieser gefüllt ist. Bei großen Speicherbereichen führt dies zu einer hohen Auslastung der CPU mit Speicherzugriffen. Währenddessen können keine anderen Operationen durchgeführt werden.

Eine wesentlich performantere, wenn auch etwas komplexere Implementierung stellt der *zirkuläre Ringbuffer* dar. Dieser ist, wie der Name bereits vermuten lässt, als Ring organisiert. Dabei ist er physisch selbstverständlich weiterhin als fortlaufender Bereich von Speicheradressen angelegt, wird jedoch durch eine zusätzliche logische Ebene als „Ring“ umorganisiert, auf den fortlaufend zugegriffen werden kann. Dadurch ist es möglich auf den Inhalt zuzugreifen, ohne dass dieser verschoben werden muss. Stattdessen werden Zeiger auf die nächste zu beschreibende oder zu lesende Speicherstelle erstellt

und mit jedem Speicher- oder Schreibzugriff geändert. Abbildung 4.19 verdeutlicht dieses Prinzip. Dadurch ist jeweils nur ein Speicherzugriff und die Änderung des Zeigers nötig, obwohl der Speicher identisch zu einem klassischen Schieberegister genutzt werden kann.

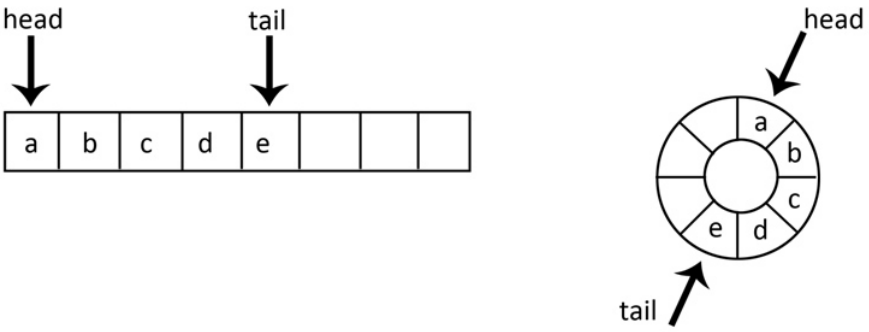



Abbildung 4.19: Prinzip des zirkulären Ringbuffers (1)[18]

Zu sehen ist, dass der belegte Speicherbereich aus einem *head* und einem *tail* besteht. Der *tail* ist dabei der Zeiger, der auf die nächste zu beschreibende oder je nach Implementierung die letzte beschriebene, Speicheradresse zeigt. Der *head* hingegen zeigt auf die nächste Adresse von der gelesen werden muss. Wird der Speicher aus Abbildung 4.19 weiter beschrieben ohne das Daten gelesen werden und er somit geleert wird, entsteht die in Abbildung 4.20 dargestellte Situation. *tail* und *head* zeigen auf die gleiche Adresse, der Speicher ist voll. Weitere Schreibvorgänge können nur durchgeführt werden, wenn zuvor aus einem Feld gelesen worden ist.

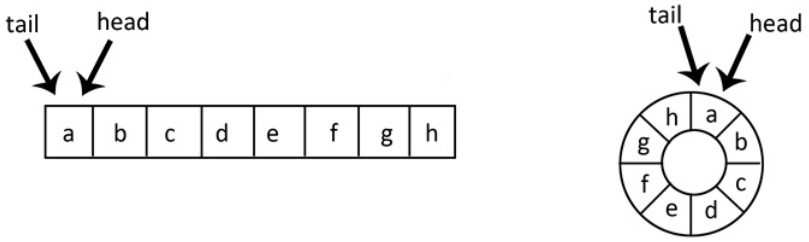



Abbildung 4.20: Prinzip des zirkulären Ringbuffers (2)[19]

In Abbildung 4.21 ist zu sehen, wie der Speicher mit neuen Daten gefüllt wird. Dazu wird aus dem *head* gelesen. Dieser zeigt nun auf die nächste Adresse. Anschließend kann in den *tail* geschrieben werden. Nachdem dieser Zeiger ebenfalls erhöht wurde, zeigen beide Pointer wieder auf die gleiche Adresse. Während diesem Vorgang rotieren die Zeiger scheinbar in einem ringförmigen Speicher, obwohl es sich physisch um einen klassisch linear organisierten handelt. Diese logische Abstraktion ist durch eine Bibliothek zu leisten. Die hierfür benötigten Funktionen werden bei dieser Anwendung durch die dafür entwickelte Sammlung bereitgestellt (Abschnitt 4.3.2).

[18] [13]
[19] [13]

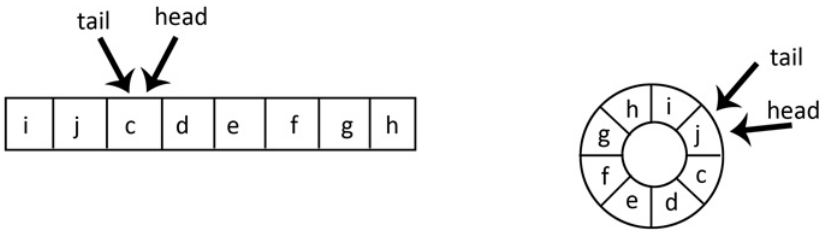



Abbildung 4.21: Prinzip des zirkulären Ringbuffers (3)[20]

### Synchronisation

Die Synchronisation stellt den wesentlichen Unterschied zur Firmware des Senders dar. Während dieser Phase muss der Synchronimpuls im Informationssignal erkannt werden, um Sender und Empfänger zeitlich abzustimmen. Dies ist nötig, damit die Entschlüsselung gelingt. Für die Detektion des Sinussignals wird die digitalisierte Information mit einem schmalbandigem, digitalen FIR-Filter bearbeitet und das Ergebnis anschließend quantitativ bewertet.

Sobald die Periodendauer für die Abtastung des kontinuierlichen Analogsignals erreicht ist, wird die ISR des Timers aufgerufen. Nachdem dieser zurückgesetzt wurde, beginnt die Verarbeitung der Daten (Quellcode 4.39, Zeile 277 ff.).

Währen der Abarbeitung der ISR des Timers, welcher als Zeitbasis für das Einlesen der Digitalwerte dient, werden die Daten dem Filter direkt zugeführt. Um das gefilterte Ergebnis besser bewerten zu können, wird der Betrag des Ausgangs gebildet und in einem Buffer gespeichert. Anschließend wird der Mittelwert über die letzten 315 Werte gebildet um das Signal zu glätten und kurzzeitige Störungen zu minimieren (Quellcode 4.39, Zeile 297 ff.).

Diese werden dazu über den SSI-Port des LCP1767 mit Hilfe der Funktion *RecSPI()* vom externen ADC angefordert. Anschließend wird der Wert in das Q15-Format konvertiert. Dies geschieht gemäß Gleichung 3.33. Dabei wird der unipolare Wert durch Subtraktion des halben Wertebereiches in eine vorzeichenbehaftete Größe umgewandelt. Im Anschluss wird er um die fünf fehlenden Bits nach links geschoben. Dies ist nötig, da Q15 16 Bits benötigt, der ADC jedoch nur 10Bit-Werte liefert. Durch die Subtraktion entsteht somit das höchstwertige Bit für das Zweierkomplement und durch die Verschiebung die Anpassung an die höher Auflösung (Quellcode 4.39, Zeile 284 ff.).

Nachdem die Nutzdaten nun im richtigen Format vorliegen, können diese gefiltert werden. Dazu wird der Filter auf jedes einzelne Sample angewendet (Abbildung 4.22). Die Zwischenergebnisse der jeweiligen Taps werden intern in der State-Struktur der Filterinstanz gespeichert, das Endergebnis wird in eine vorgegebene Variable geschrieben.

[20] [13]

```
277 void TIMER1_IRQHandler(void) {
278   NVIC_DisableIRQ(TIMER1_IRQn);
279   LPC_TIM1->MR0 = 1828;                          // 16000 Hz SampleRate
280   LPC_TIM1->TCR |= (1<<1);
281   LPC_TIM1->IR  |= (1<<0);                       // Clear MR0 interrupt flag
282   LPC_TIM1->TCR  = (1<<0);
283
284   LPC_DAC->DACR = (DACOutputVal<<6);
285
286   ADCInputVal = RecSPI();
287
288   if (decrypt_counter == BUFFERSIZE) {
289     __nop();
290   }
291
292   //  Filter start  //
293   q15_t InputQNum = (ADCInputVal-512)<<5;
294
295   q15_t OutputQNum = 0;
296
297   arm_fir_fast_q15(&FIR_BP, &InputQNum, &OutputQNum, block_size);// process
        filter
298
299   arm_abs_q15(&OutputQNum,&OutputQNum,1);
300
301   if(!BufferIn_q15_t(&OutputBufferLast,OutputQNum)) {
302     NVIC_EnableIRQ(TIMER1_IRQn);
303     return;
304   }
305
306   arm_mean_q15(OutputBufferLast.data,filter_length,&filter_output_mean);
```

Quellcode 4.39: Filterung des Eingangssignals

Nach der Betragsbildung wird das Ergebnis in einen Zwischenspeicher überführt, welcher die Werte der letzten 315 Samples enthält. Über die Werte dieses Speichers wird nun der Mittelwert gebildet (Abbildung 4.23).

Da der Filter eine Verzögerung zwischen den Signalwegen des Synchronisations- und des Nutzsignales hervorruft, muss diese zusätzlich ausgeglichen werden. Nur dadurch ist es möglich, den Datenstrom des Nutzsignales zu synchronisieren, sobald der Impuls detektiert wurde. Dazu werden die Daten des Nutzsignals durch das Schieberegister *delay_buffer* verzögert (Quellcode 4.40). Dieses besitzt prinzipiell eine feste Länge, wodurch der Laufzeitunterschied exakt ausgeglichen werden kann. Dies ist jedoch nicht immer der Fall. An der entsprechenden Stelle im Quellcode wird genauer darauf eingegangen und sowohl die Ursache für die Laufzeitunterschiede, als auch eine Lösung dafür aufgezeigt.

Die eigentliche Synchronisation wird durch eine quantitative Bewertung des Filterausgangs realisiert. Sobald der Pegel des Ausgangssignals einen gewissen Schwellwert überschreitet, wird ein entsprechendes Flag gesetzt. Fällt der Pegel anschließend unter

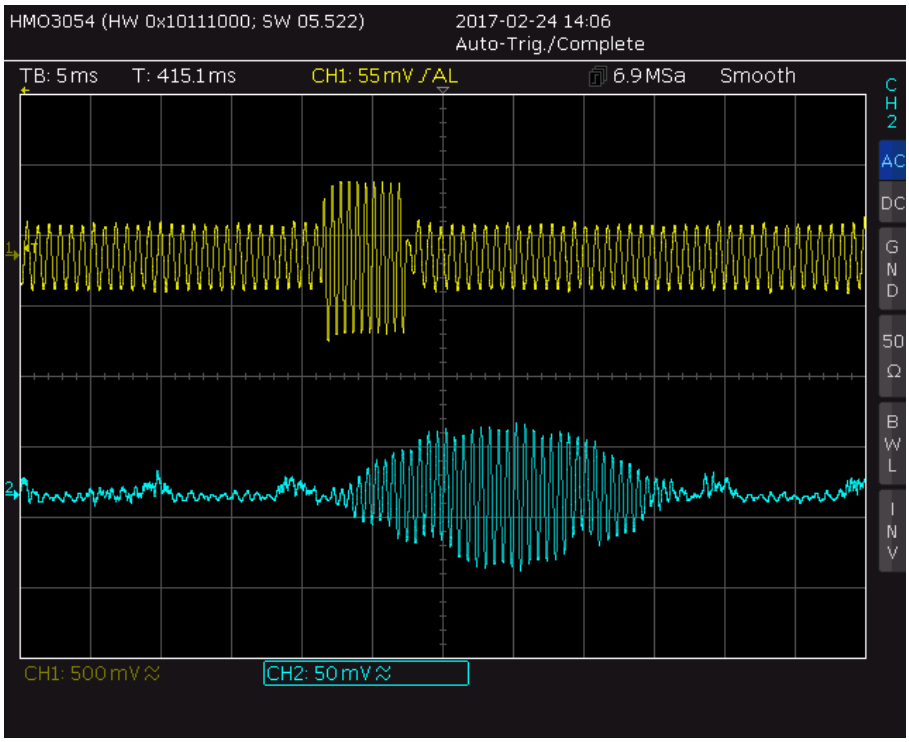



Abbildung 4.22: Unbearbeitetes Ausgangssignal des Filters

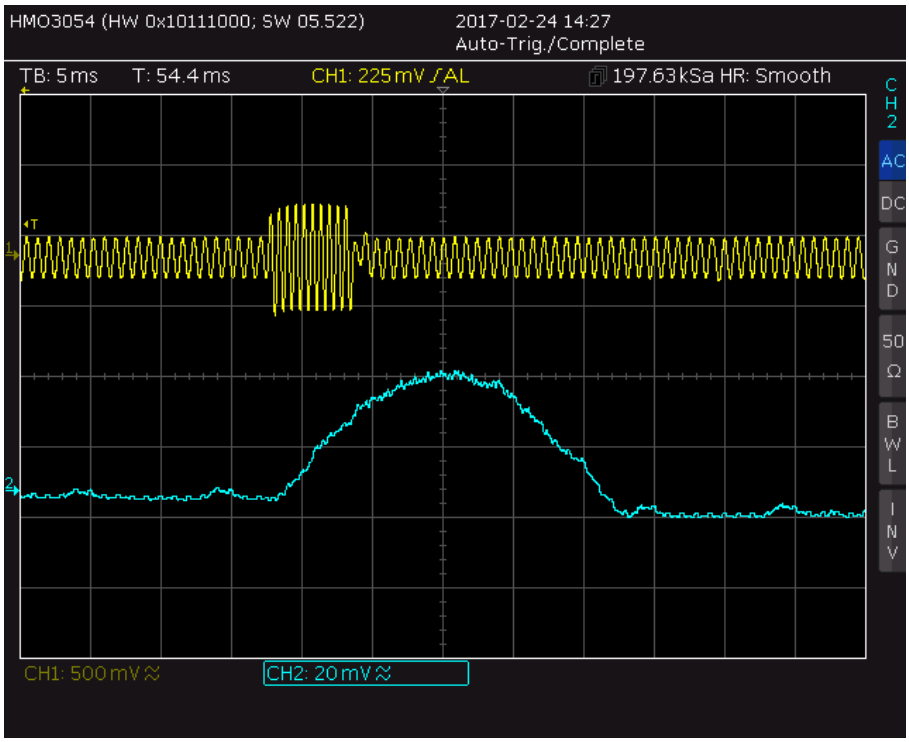



Abbildung 4.23: Unipolares, geglättetes Ausgangssignal des Filters

den Grenzwert, wird das Flag gelöscht (Quellcode 4.41, Zeile 369 ff.).

Die Samples, während denen der Synchronimpuls präsent war, werden gezählt. War der

```
309   //Data througput//
310   //move input to buffer, if encrypted decrypt it
311
312   if(!BufferIn_uint8_t(&delay_buffer,(ADCInputVal>>2))) {
313     NVIC_EnableIRQ(TIMER1_IRQn);
314     return;
315   }
316
317   temp = 128;
318
319   if (delay_buffer.data_size+1 >= delay_buffer.buffer_size) {
320     BufferOut_uint8_t(&delay_buffer,&temp);
321   }
322   else {
323     temp = 123;
324   }
```

Quellcode 4.40: Verzögerung des Nutzsignals

```
366   if ( sync_valid > 0 ) {
367     sync_valid-=1;
368   }
369   else if ( !sync_high && filter_output_mean > 300 ) {
370           if (((LPC_GPIO0->FIOPIN)>>23)&0x01) {
371       LPC_GPIO0->FIOCLR |= (1<<23);
372     }
373     else {
374       LPC_GPIO0->FIOSET |= (1<<23);
375     }
376
377     sync_counter = 0;
378     sync_high = true;
379   }
380   else if ( sync_high && filter_output_mean < 300) {
381     if (((LPC_GPIO0->FIOPIN)>>23)&0x01) {
382       LPC_GPIO0->FIOCLR |= (1<<23);
383     }
384     else {
385       LPC_GPIO0->FIOSET |= (1<<23);
386     }
387     sync_high = false;
388     sync = true;
389     if(!BufferIn_uint16_t(&sync_counter_buffer,sync_counter)) {
390       NVIC_EnableIRQ(TIMER1_IRQn);
391       return;
392     }
393   }
```

Quellcode 4.41: Detektion des Synchronimpulses (1)

Zeitraum lang genug, wird die Synchronisierung eingeleitet (Quellcode 4.42, Zeile 395 ff.). An dieser Stelle wird nun der Zwischenspeicher zur Verzögerung des Nutzsignales variabel an die Länge des Synchronimpulses angepasst. Wie in Abbildung 4.24 ersichtlich, ergibt sich die Verzögerung ($\Delta t$) prinzipiell aus der Gruppenlaufzeit des Detektions-

filters, des Mittelwertfilters ($t_{IR}$) und der halben Dauer des Zeitraumes, während der das Flag *sync_high* gesetzt war ($t_{syncAktiv}/2$), abzüglich der halben Dauer des Synchronimpulses. Ausgehend davon, dass es sich um einen richtig erkannten Synchronimpuls handelt, beträgt die Anzahl der Samples, um die das Signal verzögert werden muss, 414 (Gleichung 4.3). Dadurch findet die Synchronisation vor dem ersten empfangenen Symbol statt.

$$\begin{aligned} \Delta t &= \frac{t_{IR,FIR}}{2} + \frac{t_{IR,SMA}}{2} + \frac{t_{sync}}{2} + \frac{t_{syncAktiv}}{2} - t_{sync} \\ &= t_{IR} + \frac{t_{syncAktiv}}{2} - \frac{t_{sync}}{2} \\ &= 314 + 150 - 50 \\ &= 414 \end{aligned} \tag{4.3}$$

Weil die berechnete Dauer des Impulses und somit die Verzögerung schwanken, wird diese über die letzten fünf Werte gemittelt (Quellcode 4.42, Zeile 411 ff.). Dieser Mittelwert wird zur Berechnung der neuen Verzögerung herangezogen. Dies hat den Vorteil, dass kurzzeitige transiente Störungen gedämpft werden und sich der Filter automatisch an übertragungsabhängige Pegelschwankungen anpassen kann, solange der Pegel über dem minimalen Schwellwert bleibt. Je größer die Differenz zwischen der maximalen Amplitude am Filterausgang und dem Schwellwert dabei ist, desto stabiler ist die Erkennung des Impulses.

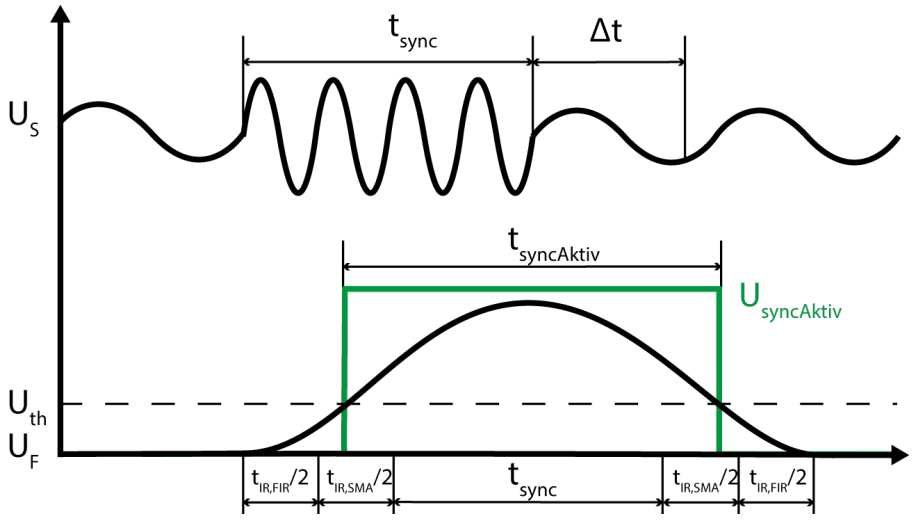



Abbildung 4.24: Zeitliche Struktur der Verzögerung

Um einen Datenverlust zu vermeiden, wird der Verzögerungsbuffer zwischengespeichert und anschließend mit der neuen Länge angelegt (Quellcode 4.42, Zeile 405 ff.). Da der Speicher als zirkulärer Ringbuffer organisiert ist, muss die dazugehörige Datenstruktur manuell aktualisiert werden.

Abschließend werden die entschlüsselten Daten in den Ausgangsbuffer des DAC ko-

```
394   if (sync_high) {
395     sync_counter++;
396   }
397   if (decrypt_counter < BUFFERSIZE-1) {
398     decrypt_counter++;
399   }
400
401   mean_prev = filter_output_mean;
402
403   if (sync && sync_valid == 0) {
404
405     unsigned char temp[delay_buffer.buffer_size];
406
407     memcpy(temp,delay_buffer.data,delay_buffer.buffer_size);
408
409     sync_mean = 0;
410     for(int i=0;i<5;i++) {
411         sync_mean += *(sync_counter_buffer.data+i);
412     }
413     sync_mean /= 5;
414
415     int new_size = FILTERSIZE+(sync_mean>>1)-(SYNC_SIZE>>1);
416     int read_tmp = delay_buffer.read+1 < new_size ? delay_buffer.read : 1;
417     int write_tmp = delay_buffer.write+1 < new_size ? delay_buffer.write :
      0;
418
419     //free & malloc new buffer with adjusted delay
420     free(delay_buffer.data);
421     delay_buffer.data = malloc((new_size)*sizeof(unsigned char));
422     delay_buffer.buffer_size = new_size;
423     delay_buffer.read = read_tmp;
424     delay_buffer.write = write_tmp;
425     delay_buffer.data_size = delay_buffer.buffer_size-(delay_buffer.read-
      delay_buffer.write)+1;
426
427     memcpy(delay_buffer.data,temp,new_size < sizeof(temp) ? new_size :
      sizeof(temp));
428
429     if (delay_buffer.data_size+1 >= delay_buffer.buffer_size) {
430       BufferOut_uint8_t(&delay_buffer,NULL);
431     }
```

Quellcode 4.42: Detektion des Synchronimpulses (2)

piert. Außerdem werden die Flags für die Synchronisation gesetzt. Das Signalflag zum Ausführen der Synchronisation wird gelöscht und *sync_valid* auf den Wert 600 gesetzt. Dies verhindert eine erneute zeitnahe Synchronisation (Quellcode 4.43).

### Entschlüsselung und Ausgabe

Nachdem Nutzsignal und Synchronimpuls nun phasengleich sind, kann der Datenstrom entschlüsselt werden, sofern dies nötig ist. Dazu wird der, während der Initialisierung

```
432     //copy decrypted data to output buffer
433     memset(dacoutbuff.data,123,dacoutbuff.buffer_size);
434     memcpy(dacoutbuff.data,decryptbuffer,BUFFERSIZE);
435     memset(decryptbuffer,123,BUFFERSIZE);
436     //reset out buffer (initialized as full, because data was copied from
        decrypt-buffer
437     dacoutbuff.read=0;
438     dacoutbuff.write=BUFFERSIZE;
439     dacoutbuff.data_size=BUFFERSIZE;
440
441     //reset counter for decrypt-buffer
442     decrypt_counter = 2;
443     sync=false;
444     sync_valid = 600;
445   }
446
447     //Remove oldest Value from Memory if not done yet
448
449   if (OutputBufferLast.data_size+1 >= OutputBufferLast.buffer_size) {
450     BufferOut_q15_t(&OutputBufferLast,NULL);
451   }
452
453   if (sync_counter_buffer.data_size+1 >= sync_counter_buffer.buffer_size) {
454     BufferOut_uint16_t(&sync_counter_buffer,NULL);
455   }
456
457   NVIC_EnableIRQ(TIMER1_IRQn);
458 };
```

Quellcode 4.43: Detektion des Synchronimpulses (3)

```
326   if (decrypt_counter > 40 && (decrypt_counter < sync_start ||
        decrypt_counter > (sync_start+sync_length))) {
327     if (keysize>0) {
328       decryptbuffer[(current_words[decrypt_counter/BLOCKSIZE]*BLOCKSIZE)+
        decrypt_counter%BLOCKSIZE] = temp;
329     }
330     else {
331       decryptbuffer[decrypt_counter] = temp;
332     }
333   }
334   else {
335     if (keysize>0) {
336       decryptbuffer[(current_words[decrypt_counter/BLOCKSIZE]*BLOCKSIZE)+
        decrypt_counter%BLOCKSIZE] = 123;
337       }
338     else {
339       decryptbuffer[decrypt_counter] = 123;
340     }
341   }
```

Quellcode 4.44: Entschlüsselung der Nutzdaten

berechnete, invertierte Kryptoschlüssel verwendet. Mit dessen Hilfe kann die ursprüngliche Position des aktuellen Samples bestimmt werden. An zwei Stellen im Speicher

findet keine Entschlüsselung statt. Dies betrifft zum einen die Positionen, an denen der Synchronimpuls gespeichert ist und zum anderen den Beginn des Datenblocks (Quellcode 4.44, Zeile 327 ff.). Die Daten werden an diesen Stellen durch den Gleichanteil des Audiosignals ersetzt, um den Sinuston aus diesem zu löschen. Der Synchronimpuls ist nun nicht mehr hörbar (Quellcode 4.44, Zeile 334 ff.). Die dadurch entstandene „Lücke“ im Signal ist aufgrund ihrer kurzen Dauer für den durchschnittlichen Hörer nicht, oder nur sehr schwer hörbar (Vgl. Abschnitt 2.6). Soll oder muss der Datenstrom nicht entschlüsselt werden, so wird dieser unverändert in den Speicher *decryptbuffer* übernommen (Quellcode 4.44, Zeile 331).

```
343   //move data to output (including delay to compensate the syncruntime)
344   if (dacoutbuff.data_size > 0) {
345     if(!BufferOut_uint8_t(&dacoutbuff,&temp)) {
346       NVIC_EnableIRQ(TIMER1_IRQn);
347       return;
348     }
349   }
350   else {
351     temp = 123;
352   }
353
354   DACOutputVal = temp<<2;
```

Quellcode 4.45: Ausgabe des Audiosignals

Die im Klartext vorliegenden Informationen können nun über den DAC ausgegeben werden. Sind Daten im entsprechenden Zwischenspeicher vorhanden, so werden sie in eine temporäre Variable geladen. Die eigentliche Ausgabe findet beim nächsten Aufruf der ISR, direkt zu Beginn statt (Quellcode 4.39, Zeile 284). Sollten noch keine Daten im Buffer vorhanden sein, so wird auch hier eine Gleichspannung vorgegeben, die der halben Referenzspannung entspricht. Dieser Wert wurde gewählt um bei auftretenden Fehlern und beim Start der Anwendung eine abrupte Änderung des Audiopegels zu vermeiden, da diese durch das sehr breite Spektrum zu stark wahrnehmbaren Störungen führt (Quellcode 4.45, Zeile 344 ff.).

Diese prozeduralen Anweisungen werden für jedes einzelne Sample abgearbeitet. Die Software ist somit in der Lage Änderungen des Signals während der Laufzeit zu erkennen und auszuwerten.

## 4.4 Konfigurationssoftware

Zur Konfiguration der Geräte, die über die Voreinstellungen der Firmware hinausgeht wurde eine entsprechende Desktopanwendung für Windows entwickelt. Diese ist in C# geschrieben und wurde mit Visual Studio 2015 erstellt. Sie dient dem Empfang und Versand der Kryptographieschlüssel via Ethernet.

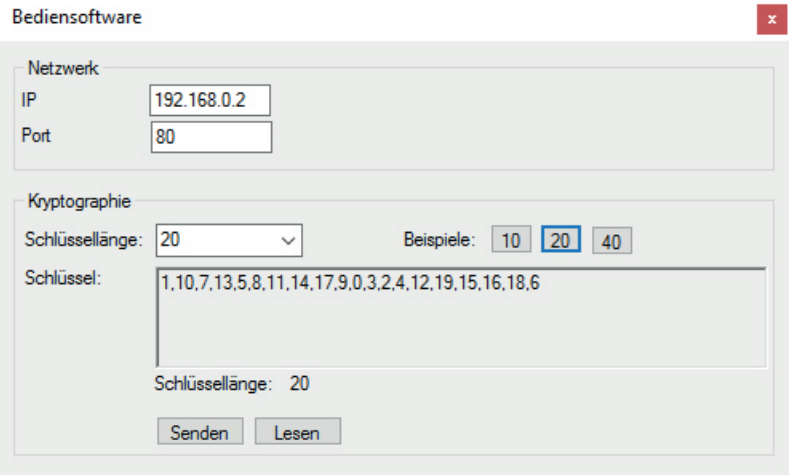



Abbildung 4.25: Abtastung eines kontinuierlichen Signals

Die Anwendung besitzt dafür zwei Bereiche. Der obere dient der Netzwerkkonfiguration. Für eine erfolgreiche Kommunikation müssen die in der Firmware definierten Werte für den Netzwerksocket eingetragen werden. Im unteren Bereich können die vordefinierten Schlüssel geladen, oder benutzerdefinierte Schlüssel eingegeben werden. Mit den entsprechenden Schaltflächen können die Daten an die Geräte übertragen, oder von ihnen abgerufen werden.

Sobald eine der Schaltflächen betätigt wird, wird das entsprechende Datenpaket gemäß dem Protokoll (Tabelle 4.5 und Tabelle 4.6) vorbereitet. (Quellcode 4.46). Anschließend werden die Daten über einen asynchronen Socket übertragen (Quellcode 4.48). Zur Verifikation der Übertragung, sendet die Firmware die Daten an die Konfigurationssoftware zurück. Diese wertet die Pakete aus und stellt den Inhalt in der grafischen Oberfläche dar (Quellcode 4.47). Der Nutzer hat so die Möglichkeit nachzuvollziehen, ob die Daten erfolgreich vom Empfänger übernommen wurden.

```
236     byte[] command = new byte[42];
237
238     command[0] = (byte)'s';
239
240     command[1] = (byte)(Convert.ToInt32(comboBox1.SelectedItem.ToString()));
241
242     for (int i = 0; i < Convert.ToInt32(comboBox1.SelectedItem);i++)
243     {
244         command[i + 2] = (byte)(Convert.ToInt32(richTextBox1.Text.Split(',')
        [i]));
245     }
```

Quellcode 4.46: Synthese des Datenpaketes

```
252     int keysize = Convert.ToInt32(resp[1]);
253
254     string[] key = new string[keysize];
255     for(int i=0;i< keysize;i++)
256     {
257         key[i] = Convert.ToString(resp[i + 2]);
258     }
259     string key_s = String.Join(",", key);
260     richTextBox1.Text = String.Join(",",key);
```

Quellcode 4.47: Analyse des Datenpaketes

```
30      connectDone.Reset();
31      sendDone.Reset();
32      receiveDone.Reset();
33
34      IPEndPoint remoteEP = new IPEndPoint(ip, port);
35
36      // create socket
37      Socket client = new Socket(AddressFamily.InterNetwork,
38      SocketType.Stream, ProtocolType.Tcp);
39      client.SendTimeout = 5;
40      client.ReceiveTimeout = 5;
41
42      // connect to endpoint
43      var result = client.BeginConnect(remoteEP,
44      new AsyncCallback(ConnectCallback), client);
45
46      var success = result.AsyncWaitHandle.WaitOne(TimeSpan.FromSeconds(1));
47
48      if (!success)
49      {
50          MessageBox.Show("Ungueltiger Socket. IP und Port ueberpruefen.", "
        Fehler", MessageBoxButtons.OK, MessageBoxIcon.Error);
51          return Encoding.Unicode.GetBytes(" Error");
52      }
53
54      connectDone.WaitOne();
55
56      // send data
57      Send(client, data);
58      sendDone.WaitOne();
59
60      // receive data
61      Receive(client);
62      receiveDone.WaitOne();
63
64      StringBuilder resp_string = new StringBuilder();
65
66
67      // decode response
68      resp_string.Append(Convert.ToChar(response[0]).ToString());
69
70      for (int i=1;i<response.Length;i++)
71      {
72          resp_string.Append(Convert.ToInt32(response[i]));
73      }
74
75      // close socket
76      client.Shutdown(SocketShutdown.Both);
77      client.Close();
```

Quellcode 4.48: Asynchroner Socket für die Datenübertragung

# 5 Ergebnisse

Die Verwendung des in Kapitel 4 beschriebenen Versuchsaufbaus zeigte bei der Inbetriebnahme die erwartete Funktion. Die Firmware ermöglicht eine stabile Synchronisation des analogen Kommunikationskanals. Abbildung 5.1 zeigt die Aufnahme eines Oszilloskops auf dem das Eingangs- und das Ausgangssignal zu sehen sind. Wie zu erkennen, wird das Sinussignal, welches der Synchronisation dient, erkannt und aus dem Informationssignal entfernt. Dies ermöglicht eine durchgehende Übertragung von Audiosignalen.

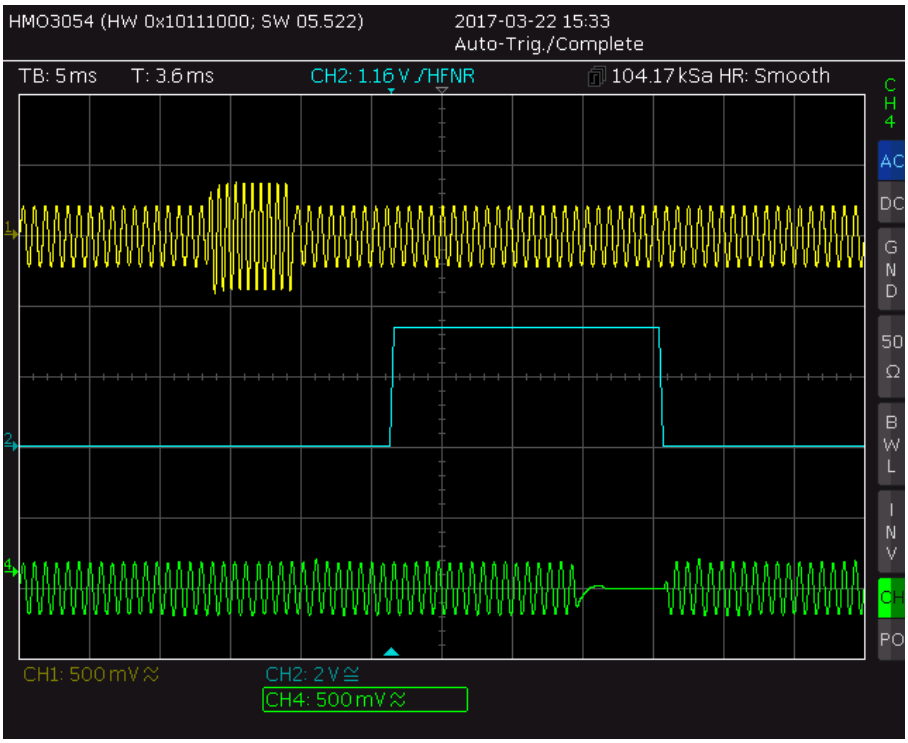



Abbildung 5.1: Ein- und Ausgangssignal des Empfängers mit Debugsignal der Synchronisation

## 5.1 Qualität der Audioübertragung

Zur Bewertung der Qualität der Übertragung wurden einige Aufnahmen mit verschiedenen Schlüssellängen für die Verschlüsselung erstellt. Wie einleitend erwähnt, ergibt sich durch die exemplarisch implementierte Verschlüsselung automatisch die Möglichkeit, die Qualität der Übertragung subjektiv bewerten zu können. Da die Verschlüsselung das Audiosignal in einzelne Symbole unterteilt und vertauscht, wird das ursprüngliche Signal bei der Entschlüsselung nur fehlerfrei rekonstruiert, wenn exakt beim ersten Sample des Blocks synchronisiert wurde. Weicht der Zeitpunkt davon ab, ergeben sich Fehler im Signal, welche je nach Ausprägung der Verschiebung hörbar werden.

Anlage C enthält die in Tabelle 5.1 aufgeführten Audiodateien, welche zur subjektiven Bewertung der Übertragungsqualität herangezogen wurden.
Die Übertragung ohne Verschlüsselung bietet eine für analoge Telefongespräche übliche Qualität. Der Inhalt ist klar verständlich. Durch die Begrenzung der Bandbreite gehen jedoch Signalanteile verloren, welches deutlich Hörbar ist. Die Aufnahme klingt „blechern“ und weniger dynamisch als das Original. Je größer die Anzahl der verwendeten Symbole wird, desto stärker wird der Fehler bei der Entschlüsselung, welcher durch die Abweichung bei der Synchronisation hervorgerufen wird, deutlich. Bei einer Schlüssellänge von 40 Symbolen ist bereits eine deutliche Verzerrung wahrzunehmen. Der Inhalt ist zwar immer noch verständlich, jedoch bereits hörbar fehlerbehaftet. Eine weitere Erhöhung der Symbolanzahl würde entsprechend dazu führen, dass der Inhalt der Übertragung zunehmend weniger und schließlich gar nicht mehr verständlich wäre.

| **Hörprobe** | **Dateiname** | | **Schlüssellänge** |
|---|---|---|---|
| | klar | verschlüsselt | |
| 1 | plain.wav | - | - |
| 2 | hoerprobe1.wav | hoerprobe1_enc.wav | 10 |
| 3 | hoerprobe2.wav | hoerprobe2_enc.wav | 20 |
| 4 | hoerprobe3.wav | hoerprobe3_enc.wav | 40 |

Tabelle 5.1: Hörproben der Übertragung

# 6 Fazit und Ausblick

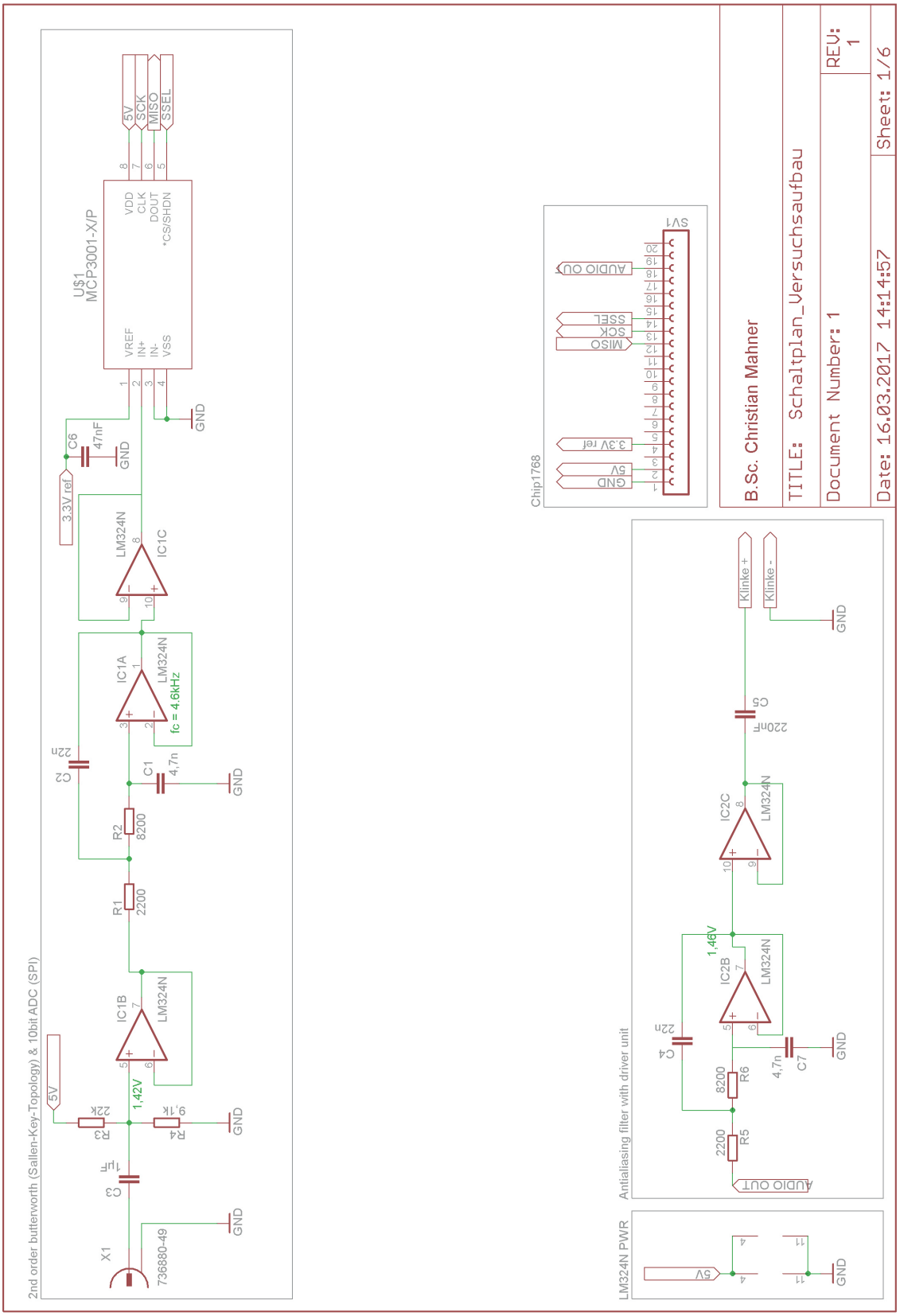
Die Aufgabe dieser Abschlussarbeit bestand darin ein Verfahren zur Synchronisation analoger Kommunikationskanäle zu entwickeln. Dabei sollten sowohl die Möglichkeiten untersucht werden, wie dieses mit den in Abschnitt 1.1 gestellten Forderungen realisiert werden kann, als auch ein entsprechender Laborversuch aufgebaut werden. In Kapitel 3 wurden verschiedene Verfahren aufgezeigt, welche dafür in Frage kommen. Das aussichtsreichste wurde schließlich favorisiert und entsprechend den Vorgaben realisiert (Kapitel 4). In Anbetracht der zur Verfügung stehenden Möglichkeiten bezüglich Hardware, technischer Ausstattung und Materialien, sowie dem zeitlichen Rahmen, kann die Aufgabenstellung als erfüllt betrachtet werden. Wie in Kapitel 5 beschrieben, hält die Applikation alle Anforderungen ein und liefert das gewünschte Ergebnis. Dennoch besteht Potential für weitere Verbesserungen bei Firm- und Hardware. Sollten weitere Untersuchungen und Entwicklungen durchgeführt werden, welche auf dieser Arbeit aufbauen, sollte dieses zuvor ausgeschöpft werden um die bestmögliche Übertragungsqualität gewährleisten zu können.

Einige Punkte, welche dazu beitragen würden, sollen abschließend kurz erläutert werden, da diese während der Bearbeitung des Projektes zu Einschränkungen aufgrund nötiger Kompromisse bezüglich der Leistungsfähigkeit geführt haben.
Zum einen sollte die Wahl des Controllers, oder zumindest die des Derivats, überdacht werden, da dieser nicht für diese Art der Anwendung ausgelegt ist und der Arbeitsspeicher des Controllers nicht erweitert werden kann. Ein schnellerer Controller, beziehungsweise ein Modell mit entsprechender Gleitkommaeinheit, würde eine höhere Abtastrate erlauben und somit die Qualität des Audiosignals verbessern. Die Erweiterung des Speichers oder ein Controller mit mehr integriertem Speicher wäre für die Auflösung der digitalisierten Daten und des Filters von Vorteil, da beide durch die Quantisierung auf eine verhältnismäßig geringe Auflösung verfälscht werden. Auch diese hätte positive Einflüsse auf die Qualität des Audiosignals und des Detektionsfilters.
Abseits des Controllers selbst könnte eine Erhöhung der Filterordnung des Anti-Aliasing-Filters und des Rekonstruktionsfilters das Signal verbessern und Störungen vermindern.
Ein weiterer Faktor der zur Verschlechterung der Qualität der Übertragung führte, sind die Ausführung der Chinch-Leitungen. Bei diesen handelt es sich um minderwertige Steckverbinder und ungeschirmte Leitungen, sodass diese äußerst anfällig für die Kopplung von Störungen sind. Qualitativ hochwertigere Leitungen und Steckverbinder würden auf der einen Seite Störungen vermindern und auf der anderen die Fehlersuche und Störungsanalyse während zukünftigen Entwicklungen vereinfachen, da dadurch weniger zusätzliche, stochastische Störungen eingetragen werden und das System besser analysiert werden kann.
Auch der Entwurf und der Einsatz einer gedruckten Leiterplatte können dazu beitragen die Qualität deutlich zu verbessern. Kürzere, topologieoptimierte und mechanisch

beanspruchbare Leiterzüge sowie besser Möglichkeiten zur EMV-gerechten Gestaltung bieten deutliche Vorteile gegenüber einem handgelötetem Versuchsaufbau auf Lochrasterplatine.

Trotz dieser Mängel bzw. in Anbetracht des Ergebnisses, gerade deswegen, überzeugt die Idee mit ihrer Robustheit und dem geringen technischem Aufwand. Darüber hinaus erzielt das System auch in der aktuellen Entwicklungsstufe bereits Ergebnisse, welche die Anforderungen voll und ganz erfüllen. Vor allem die zuverlässige und verhältnismäßig effiziente Detektion des Synchronimpulses sticht heraus. Obwohl es heutzutage wesentlich modernere und in vielen Bereichen überlegenere Konkurrenzverfahren gibt, kann dieses für vereinzelte Anwendungen, die nicht unbedingt der Audioübertragung dienen müssen, eine Alternative darstellen. Und auch wenn dies keine Entwicklung für den Massenmarkt darstellen wird, so konnte zumindest gezeigt werden, dass auch althergebrachte Verfahren mit moderner Technik und Software durchaus zufriedenstellende Ergebnisse liefern können.

# Anhang A: Schaltplan



# Anhang B: Filterkoeffizienten

Skalierte, diskrete Filterkoeffizienten $b_w[k]$:

$$
\begin{aligned}
b_w[k] = \{ \quad & -9.270\text{e}-04, \quad -2.465\text{e}-03, \quad -3.092\text{e}-03, \quad -2.539\text{e}-03, \\
& -9.840\text{e}-04, \quad 9.980\text{e}-04, \quad 2.651\text{e}-03, \quad 3.323\text{e}-03, \\
& 2.726\text{e}-03, \quad 1.056\text{e}-03, \quad -1.070\text{e}-03, \quad -2.840\text{e}-03, \\
& -3.558\text{e}-03, \quad -2.917\text{e}-03, \quad -1.129\text{e}-03, \quad 1.144\text{e}-03, \\
& 3.033\text{e}-03, \quad 3.797\text{e}-03, \quad 3.111\text{e}-03, \quad 1.203\text{e}-03, \\
& -1.218\text{e}-03, \quad -3.228\text{e}-03, \quad -4.039\text{e}-03, \quad -3.307\text{e}-03, \\
& -1.278\text{e}-03, \quad 1.293\text{e}-03, \quad 3.425\text{e}-03, \quad 4.282\text{e}-03, \\
& 3.504\text{e}-03, \quad 1.353\text{e}-03, \quad -1.369\text{e}-03, \quad -3.622\text{e}-03, \\
& -4.526\text{e}-03, \quad -3.702\text{e}-03, \quad -1.429\text{e}-03, \quad 1.444\text{e}-03, \\
& 3.820\text{e}-03, \quad 4.771\text{e}-03, \quad 3.899\text{e}-03, \quad 1.504\text{e}-03, \\
& -1.520\text{e}-03, \quad -4.018\text{e}-03, \quad -5.015\text{e}-03, \quad -4.096\text{e}-03, \\
& -1.580\text{e}-03, \quad 1.595\text{e}-03, \quad 4.214\text{e}-03, \quad 5.257\text{e}-03, \\
& 4.292\text{e}-03, \quad 1.654\text{e}-03, \quad -1.669\text{e}-03, \quad -4.408\text{e}-03, \\
& -5.497\text{e}-03, \quad -4.485\text{e}-03, \quad -1.728\text{e}-03, \quad 1.742\text{e}-03, \\
& 4.600\text{e}-03, \quad 5.733\text{e}-03, \quad 4.676\text{e}-03, \quad 1.800\text{e}-03, \\
& -1.815\text{e}-03, \quad -4.788\text{e}-03, \quad -5.965\text{e}-03, \quad -4.862\text{e}-03, \\
& -1.871\text{e}-03, \quad 1.885\text{e}-03, \quad 4.972\text{e}-03, \quad 6.191\text{e}-03, \\
& 5.045\text{e}-03, \quad 1.941\text{e}-03, \quad -1.954\text{e}-03, \quad -5.152\text{e}-03, \\
& -6.411\text{e}-03, \quad -5.222\text{e}-03, \quad -2.008\text{e}-03, \quad 2.021\text{e}-03, \\
& 5.325\text{e}-03, \quad 6.624\text{e}-03, \quad 5.393\text{e}-03, \quad 2.073\text{e}-03, \\
& -2.085\text{e}-03, \quad -5.493\text{e}-03, \quad -6.830\text{e}-03, \quad -5.558\text{e}-03, \\
& -2.135\text{e}-03, \quad 2.147\text{e}-03, \quad 5.653\text{e}-03, \quad 7.026\text{e}-03, \\
& 5.715\text{e}-03, \quad 2.195\text{e}-03, \quad -2.206\text{e}-03, \quad -5.806\text{e}-03, \\
& -7.213\text{e}-03, \quad -5.864\text{e}-03, \quad -2.251\text{e}-03, \quad 2.262\text{e}-03, \\
& 5.950\text{e}-03, \quad 7.389\text{e}-03, \quad 6.005\text{e}-03, \quad 2.304\text{e}-03, \\
& -2.314\text{e}-03, \quad -6.085\text{e}-03, \quad -7.554\text{e}-03, \quad -6.137\text{e}-03, \\
& -2.354\text{e}-03, \quad 2.363\text{e}-03, \quad 6.211\text{e}-03, \quad 7.707\text{e}-03, \\
& 6.259\text{e}-03, \quad 2.400\text{e}-03, \quad -2.408\text{e}-03, \quad -6.327\text{e}-03, \\
& -7.848\text{e}-03, \quad -6.371\text{e}-03, \quad -2.441\text{e}-03, \quad 2.449\text{e}-03, \\
& 6.433\text{e}-03, \quad 7.976\text{e}-03, \quad 6.472\text{e}-03, \quad 2.479\text{e}-03, \\
& -2.486\text{e}-03, \quad -6.527\text{e}-03, \quad -8.090\text{e}-03, \quad -6.562\text{e}-03, \\
& -2.513\text{e}-03, \quad 2.519\text{e}-03, \quad 6.610\text{e}-03, \quad 8.190\text{e}-03, \\
& 6.640\text{e}-03, \quad 2.542\text{e}-03, \quad -2.547\text{e}-03, \quad -6.682\text{e}-03, \\
& -8.275\text{e}-03, \quad -6.707\text{e}-03, \quad -2.566\text{e}-03, \quad 2.571\text{e}-03, \\
& 6.741\text{e}-03, \quad 8.345\text{e}-03, \quad 6.761\text{e}-03, \quad 2.586\text{e}-03, \\
& -2.589\text{e}-03, \quad -6.788\text{e}-03, \quad -8.400\text{e}-03, \quad -6.803\text{e}-03, \\
& -2.601\text{e}-03, \quad 2.604\text{e}-03, \quad 6.822\text{e}-03, \quad 8.439\text{e}-03, \\
& 6.832\text{e}-03, \quad 2.611\text{e}-03, \quad -2.613\text{e}-03, \quad -6.844\text{e}-03,
\end{aligned}
$$

$$
\begin{array}{rrrr}
-8.463\text{e}-03, & -6.849\text{e}-03, & -2.617\text{e}-03, & 2.617\text{e}-03, \\
6.853\text{e}-03, & 8.471\text{e}-03, & 6.853\text{e}-03, & 2.617\text{e}-03, \\
-2.617\text{e}-03, & -6.849\text{e}-03, & -8.463\text{e}-03, & -6.844\text{e}-03, \\
-2.613\text{e}-03, & 2.611\text{e}-03, & 6.832\text{e}-03, & 8.439\text{e}-03, \\
6.822\text{e}-03, & 2.604\text{e}-03, & -2.601\text{e}-03, & -6.803\text{e}-03, \\
-8.400\text{e}-03, & -6.788\text{e}-03, & -2.589\text{e}-03, & 2.586\text{e}-03, \\
6.761\text{e}-03, & 8.345\text{e}-03, & 6.741\text{e}-03, & 2.571\text{e}-03, \\
-2.566\text{e}-03, & -6.707\text{e}-03, & -8.275\text{e}-03, & -6.682\text{e}-03, \\
-2.547\text{e}-03, & 2.542\text{e}-03, & 6.640\text{e}-03, & 8.190\text{e}-03, \\
6.610\text{e}-03, & 2.519\text{e}-03, & -2.513\text{e}-03, & -6.562\text{e}-03, \\
-8.090\text{e}-03, & -6.527\text{e}-03, & -2.486\text{e}-03, & 2.479\text{e}-03, \\
6.472\text{e}-03, & 7.976\text{e}-03, & 6.433\text{e}-03, & 2.449\text{e}-03, \\
-2.441\text{e}-03, & -6.371\text{e}-03, & -7.848\text{e}-03, & -6.327\text{e}-03, \\
-2.408\text{e}-03, & 2.400\text{e}-03, & 6.259\text{e}-03, & 7.707\text{e}-03, \\
6.211\text{e}-03, & 2.363\text{e}-03, & -2.354\text{e}-03, & -6.137\text{e}-03, \\
-7.554\text{e}-03, & -6.085\text{e}-03, & -2.314\text{e}-03, & 2.304\text{e}-03, \\
6.005\text{e}-03, & 7.389\text{e}-03, & 5.950\text{e}-03, & 2.262\text{e}-03, \\
-2.251\text{e}-03, & -5.864\text{e}-03, & -7.213\text{e}-03, & -5.806\text{e}-03, \\
-2.206\text{e}-03, & 2.195\text{e}-03, & 5.715\text{e}-03, & 7.026\text{e}-03, \\
5.653\text{e}-03, & 2.147\text{e}-03, & -2.135\text{e}-03, & -5.558\text{e}-03, \\
-6.830\text{e}-03, & -5.493\text{e}-03, & -2.085\text{e}-03, & 2.073\text{e}-03, \\
5.393\text{e}-03, & 6.624\text{e}-03, & 5.325\text{e}-03, & 2.021\text{e}-03, \\
-2.008\text{e}-03, & -5.222\text{e}-03, & -6.411\text{e}-03, & -5.152\text{e}-03, \\
-1.954\text{e}-03, & 1.941\text{e}-03, & 5.045\text{e}-03, & 6.191\text{e}-03, \\
4.972\text{e}-03, & 1.885\text{e}-03, & -1.871\text{e}-03, & -4.862\text{e}-03, \\
-5.965\text{e}-03, & -4.788\text{e}-03, & -1.815\text{e}-03, & 1.800\text{e}-03, \\
4.676\text{e}-03, & 5.733\text{e}-03, & 4.600\text{e}-03, & 1.742\text{e}-03, \\
-1.728\text{e}-03, & -4.485\text{e}-03, & -5.497\text{e}-03, & -4.408\text{e}-03, \\
-1.669\text{e}-03, & 1.654\text{e}-03, & 4.292\text{e}-03, & 5.257\text{e}-03, \\
4.214\text{e}-03, & 1.595\text{e}-03, & -1.580\text{e}-03, & -4.096\text{e}-03, \\
-5.015\text{e}-03, & -4.018\text{e}-03, & -1.520\text{e}-03, & 1.504\text{e}-03, \\
3.899\text{e}-03, & 4.771\text{e}-03, & 3.820\text{e}-03, & 1.444\text{e}-03, \\
-1.429\text{e}-03, & -3.702\text{e}-03, & -4.526\text{e}-03, & -3.622\text{e}-03, \\
-1.369\text{e}-03, & 1.353\text{e}-03, & 3.504\text{e}-03, & 4.282\text{e}-03, \\
3.425\text{e}-03, & 1.293\text{e}-03, & -1.278\text{e}-03, & -3.307\text{e}-03, \\
-4.039\text{e}-03, & -3.228\text{e}-03, & -1.218\text{e}-03, & 1.203\text{e}-03, \\
3.111\text{e}-03, & 3.797\text{e}-03, & 3.033\text{e}-03, & 1.144\text{e}-03, \\
-1.129\text{e}-03, & -2.917\text{e}-03, & -3.558\text{e}-03, & -2.840\text{e}-03, \\
-1.070\text{e}-03, & 1.056\text{e}-03, & 2.726\text{e}-03, & 3.323\text{e}-03, \\
2.651\text{e}-03, & 9.980\text{e}-04, & -9.840\text{e}-04, & -2.539\text{e}-03, \\
-3.092\text{e}-03, & -2.465\text{e}-03, & -9.270\text{e}-04\}
\end{array}
$$

Skalierte, quantisierte, diskrete Filterkoeffizienten $b_w[k]$ im Format Q15:

$$
\begin{aligned}
b_w[k] = \{ \quad & -30, -81, -101, -83, -32, 33, \\
& 87, 109, 89, 35, -35, -93, \\
& -117, -96, -37, 37, 99, 124, \\
& 102, 39, -40, -106, -132, -108, \\
& -42, 42, 112, 140, 115, 44, \\
& -45, -119, -148, -121, -47, 47, \\
& 125, 156, 128, 49, -50, -132, \\
& -164, -134, -52, 52, 138, 172, \\
& 141, 54, -55, -144, -180, -147, \\
& -57, 57, 151, 188, 153, 59, \\
& -59, -157, -195, -159, -61, 62, \\
& 163, 203, 165, 64, -64, -169, \\
& -210, -171, -66, 66, 175, 217, \\
& 177, 68, -68, -180, -224, -182, \\
& -70, 70, 185, 230, 187, 72, \\
& -72, -190, -236, -192, -74, 74, \\
& 195, 242, 197, 76, -76, -199, \\
& -248, -201, -77, 77, 204, 253, \\
& 205, 79, -79, -207, -257, -209, \\
& -80, 80, 211, 261, 212, 81, \\
& -81, -214, -265, -215, -82, 83, \\
& 217, 268, 218, 83, -83, -219, \\
& -271, -220, -84, 84, 221, 273, \\
& 222, 85, -85, -222, -275, -223, \\
& -85, 85, 224, 277, 224, 86, \\
& -86, -224, -277, -224, -86, 86, \\
& 225, 278, 225, 86, -86, -224, \\
& -277, -224, -86, 86, 224, 277, \\
& 224, 85, -85, -223, -275, -222, \\
& -85, 85, 222, 273, 221, 84, \\
& -84, -220, -271, -219, -83, 83, \\
& 218, 268, 217, 83, -82, -215, \\
& -265, -214, -81, 81, 212, 261, \\
& 211, 80, -80, -209, -257, -207, \\
& -79, 79, 205, 253, 204, 77, \\
& -77, -201, -248, -199, -76, 76, \\
& 197, 242, 195, 74, -74, -192, \\
& -236, -190, -72, 72, 187, 230, \\
& 185, 70, -70, -182, -224, -180, \\
& -68, 68, 177, 217, 175, 66, \\
& -66, -171, -210, -169, -64, 64,
\end{aligned}
$$

$$
\begin{array}{rrrrrr}
165, & 203, & 163, & 62, & -61, & -159, \\
-195, & -157, & -59, & 59, & 153, & 188, \\
151, & 57, & -57, & -147, & -180, & -144, \\
-55, & 54, & 141, & 172, & 138, & 52, \\
-52, & -134, & -164, & -132, & -50, & 49, \\
128, & 156, & 125, & 47, & -47, & -121, \\
-148, & -119, & -45, & 44, & 115, & 140, \\
112, & 42, & -42, & -108, & -132, & -106, \\
-40, & 39, & 102, & 124, & 99, & 37, \\
-37, & -96, & -117, & -93, & -35, & 35, \\
89, & 109, & 87, & 33, & -32, & -83, \\
-101, & -81, & -30\}
\end{array}
$$

Quantisierte, diskrete Filterkoeffizienten $b_w[k]$:

$$
\begin{aligned}
b_w[k] = \{ \quad & -8.210\text{e}-04, \quad -2.182\text{e}-03, \quad -2.738\text{e}-03, \quad -2.248\text{e}-03, \\
& -8.710\text{e}-04, \quad 8.840\text{e}-04, \quad 2.347\text{e}-03, \quad 2.942\text{e}-03, \\
& 2.414\text{e}-03, \quad 9.350\text{e}-04, \quad -9.480\text{e}-04, \quad -2.515\text{e}-03, \\
& -3.151\text{e}-03, \quad -2.583\text{e}-03, \quad -1.000\text{e}-03, \quad 1.013\text{e}-03, \\
& 2.686\text{e}-03, \quad 3.362\text{e}-03, \quad 2.754\text{e}-03, \quad 1.065\text{e}-03, \\
& -1.078\text{e}-03, \quad -2.858\text{e}-03, \quad -3.576\text{e}-03, \quad -2.928\text{e}-03, \\
& -1.132\text{e}-03, \quad 1.145\text{e}-03, \quad 3.032\text{e}-03, \quad 3.791\text{e}-03, \\
& 3.102\text{e}-03, \quad 1.198\text{e}-03, \quad -1.212\text{e}-03, \quad -3.207\text{e}-03, \\
& -4.008\text{e}-03, \quad -3.277\text{e}-03, \quad -1.265\text{e}-03, \quad 1.279\text{e}-03, \\
& 3.382\text{e}-03, \quad 4.224\text{e}-03, \quad 3.452\text{e}-03, \quad 1.332\text{e}-03, \\
& -1.345\text{e}-03, \quad -3.557\text{e}-03, \quad -4.440\text{e}-03, \quad -3.627\text{e}-03, \\
& -1.399\text{e}-03, \quad 1.412\text{e}-03, \quad 3.731\text{e}-03, \quad 4.655\text{e}-03, \\
& 3.800\text{e}-03, \quad 1.465\text{e}-03, \quad -1.478\text{e}-03, \quad -3.903\text{e}-03, \\
& -4.867\text{e}-03, \quad -3.971\text{e}-03, \quad -1.530\text{e}-03, \quad 1.543\text{e}-03, \\
& 4.073\text{e}-03, \quad 5.076\text{e}-03, \quad 4.140\text{e}-03, \quad 1.594\text{e}-03, \\
& -1.607\text{e}-03, \quad -4.240\text{e}-03, \quad -5.281\text{e}-03, \quad -4.305\text{e}-03, \\
& -1.657\text{e}-03, \quad 1.669\text{e}-03, \quad 4.403\text{e}-03, \quad 5.482\text{e}-03, \\
& 4.467\text{e}-03, \quad 1.718\text{e}-03, \quad -1.730\text{e}-03, \quad -4.561\text{e}-03, \\
& -5.677\text{e}-03, \quad -4.623\text{e}-03, \quad -1.778\text{e}-03, \quad 1.789\text{e}-03, \\
& 4.715\text{e}-03, \quad 5.865\text{e}-03, \quad 4.775\text{e}-03, \quad 1.835\text{e}-03, \\
& -1.846\text{e}-03, \quad -4.863\text{e}-03, \quad -6.047\text{e}-03, \quad -4.921\text{e}-03, \\
& -1.890\text{e}-03, \quad 1.901\text{e}-03, \quad 5.005\text{e}-03, \quad 6.221\text{e}-03, \\
& 5.060\text{e}-03, \quad 1.943\text{e}-03, \quad -1.953\text{e}-03, \quad -5.140\text{e}-03, \\
& -6.386\text{e}-03, \quad -5.192\text{e}-03, \quad -1.993\text{e}-03, \quad 2.003\text{e}-03, \\
& 5.268\text{e}-03, \quad 6.542\text{e}-03, \quad 5.317\text{e}-03, \quad 2.040\text{e}-03, \\
& -2.049\text{e}-03, \quad -5.388\text{e}-03, \quad -6.688\text{e}-03, \quad -5.434\text{e}-03, \\
& -2.084\text{e}-03, \quad 2.092\text{e}-03, \quad 5.500\text{e}-03, \quad 6.824\text{e}-03, \\
& 5.542\text{e}-03, \quad 2.125\text{e}-03, \quad -2.132\text{e}-03, \quad -5.602\text{e}-03, \\
& -6.949\text{e}-03, \quad -5.641\text{e}-03, \quad -2.162\text{e}-03, \quad 2.169\text{e}-03, \\
& 5.696\text{e}-03, \quad 7.062\text{e}-03, \quad 5.730\text{e}-03, \quad 2.195\text{e}-03, \\
& -2.201\text{e}-03, \quad -5.779\text{e}-03, \quad -7.163\text{e}-03, \quad -5.810\text{e}-03, \\
& -2.225\text{e}-03, \quad 2.230\text{e}-03, \quad 5.853\text{e}-03, \quad 7.251\text{e}-03, \\
& 5.879\text{e}-03, \quad 2.250\text{e}-03, \quad -2.255\text{e}-03, \quad -5.916\text{e}-03, \\
& -7.326\text{e}-03, \quad -5.938\text{e}-03, \quad -2.272\text{e}-03, \quad 2.276\text{e}-03, \\
& 5.968\text{e}-03, \quad 7.388\text{e}-03, \quad 5.986\text{e}-03, \quad 2.290\text{e}-03, \\
& -2.293\text{e}-03, \quad -6.010\text{e}-03, \quad -7.437\text{e}-03, \quad -6.023\text{e}-03, \\
& -2.303\text{e}-03, \quad 2.305\text{e}-03, \quad 6.040\text{e}-03, \quad 7.472\text{e}-03, \\
& 6.049\text{e}-03, \quad 2.312\text{e}-03, \quad -2.313\text{e}-03, \quad -6.059\text{e}-03, \\
& -7.493\text{e}-03, \quad -6.064\text{e}-03, \quad -2.317\text{e}-03, \quad 2.317\text{e}-03, \\
& 6.067\text{e}-03, \quad 7.500\text{e}-03, \quad 6.067\text{e}-03, \quad 2.317\text{e}-03, \\
& -2.317\text{e}-03, \quad -6.064\text{e}-03, \quad -7.493\text{e}-03, \quad -6.059\text{e}-03,
\end{aligned}
$$

$$
\begin{array}{rrrr}
-2.313\text{e}-03, & 2.312\text{e}-03, & 6.049\text{e}-03, & 7.472\text{e}-03, \\
6.040\text{e}-03, & 2.305\text{e}-03, & -2.303\text{e}-03, & -6.023\text{e}-03, \\
-7.437\text{e}-03, & -6.010\text{e}-03, & -2.293\text{e}-03, & 2.290\text{e}-03, \\
5.986\text{e}-03, & 7.388\text{e}-03, & 5.968\text{e}-03, & 2.276\text{e}-03, \\
-2.272\text{e}-03, & -5.938\text{e}-03, & -7.326\text{e}-03, & -5.916\text{e}-03, \\
-2.255\text{e}-03, & 2.250\text{e}-03, & 5.879\text{e}-03, & 7.251\text{e}-03, \\
5.853\text{e}-03, & 2.230\text{e}-03, & -2.225\text{e}-03, & -5.810\text{e}-03, \\
-7.163\text{e}-03, & -5.779\text{e}-03, & -2.201\text{e}-03, & 2.195\text{e}-03, \\
5.730\text{e}-03, & 7.062\text{e}-03, & 5.696\text{e}-03, & 2.169\text{e}-03, \\
-2.162\text{e}-03, & -5.641\text{e}-03, & -6.949\text{e}-03, & -5.602\text{e}-03, \\
-2.132\text{e}-03, & 2.125\text{e}-03, & 5.542\text{e}-03, & 6.824\text{e}-03, \\
5.500\text{e}-03, & 2.092\text{e}-03, & -2.084\text{e}-03, & -5.434\text{e}-03, \\
-6.688\text{e}-03, & -5.388\text{e}-03, & -2.049\text{e}-03, & 2.040\text{e}-03, \\
5.317\text{e}-03, & 6.542\text{e}-03, & 5.268\text{e}-03, & 2.003\text{e}-03, \\
-1.993\text{e}-03, & -5.192\text{e}-03, & -6.386\text{e}-03, & -5.140\text{e}-03, \\
-1.953\text{e}-03, & 1.943\text{e}-03, & 5.060\text{e}-03, & 6.221\text{e}-03, \\
5.005\text{e}-03, & 1.901\text{e}-03, & -1.890\text{e}-03, & -4.921\text{e}-03, \\
-6.047\text{e}-03, & -4.863\text{e}-03, & -1.846\text{e}-03, & 1.835\text{e}-03, \\
4.775\text{e}-03, & 5.865\text{e}-03, & 4.715\text{e}-03, & 1.789\text{e}-03, \\
-1.778\text{e}-03, & -4.623\text{e}-03, & -5.677\text{e}-03, & -4.561\text{e}-03, \\
-1.730\text{e}-03, & 1.718\text{e}-03, & 4.467\text{e}-03, & 5.482\text{e}-03, \\
4.403\text{e}-03, & 1.669\text{e}-03, & -1.657\text{e}-03, & -4.305\text{e}-03, \\
-5.281\text{e}-03, & -4.240\text{e}-03, & -1.607\text{e}-03, & 1.594\text{e}-03, \\
4.140\text{e}-03, & 5.076\text{e}-03, & 4.073\text{e}-03, & 1.543\text{e}-03, \\
-1.530\text{e}-03, & -3.971\text{e}-03, & -4.867\text{e}-03, & -3.903\text{e}-03, \\
-1.478\text{e}-03, & 1.465\text{e}-03, & 3.800\text{e}-03, & 4.655\text{e}-03, \\
3.731\text{e}-03, & 1.412\text{e}-03, & -1.399\text{e}-03, & -3.627\text{e}-03, \\
-4.440\text{e}-03, & -3.557\text{e}-03, & -1.345\text{e}-03, & 1.332\text{e}-03, \\
3.452\text{e}-03, & 4.224\text{e}-03, & 3.382\text{e}-03, & 1.279\text{e}-03, \\
-1.265\text{e}-03, & -3.277\text{e}-03, & -4.008\text{e}-03, & -3.207\text{e}-03, \\
-1.212\text{e}-03, & 1.198\text{e}-03, & 3.102\text{e}-03, & 3.791\text{e}-03, \\
3.032\text{e}-03, & 1.145\text{e}-03, & -1.132\text{e}-03, & -2.928\text{e}-03, \\
-3.576\text{e}-03, & -2.858\text{e}-03, & -1.078\text{e}-03, & 1.065\text{e}-03, \\
2.754\text{e}-03, & 3.362\text{e}-03, & 2.686\text{e}-03, & 1.013\text{e}-03, \\
-1.000\text{e}-03, & -2.583\text{e}-03, & -3.151\text{e}-03, & -2.515\text{e}-03, \\
-9.480\text{e}-04, & 9.350\text{e}-04, & 2.414\text{e}-03, & 2.942\text{e}-03, \\
2.347\text{e}-03, & 8.840\text{e}-04, & -8.710\text{e}-04, & -2.248\text{e}-03, \\
-2.738\text{e}-03, & -2.182\text{e}-03, & -8.210\text{e}-04\} &
\end{array}
$$

Quantisierte, diskrete Filterkoeffizienten $b_w[k]$ im Format Q15:

$$
\begin{aligned}
b_w[k] = \{ \quad & -27, -72, -90, -74, -29, 29, \\
& 77, 96, 79, 31, -31, -82, \\
& -103, -85, -33, 33, 88, 110, \\
& 90, 35, -35, -94, -117, -96, \\
& -37, 38, 99, 124, 102, 39, \\
& -40, -105, -131, -107, -41, 42, \\
& 111, 138, 113, 44, -44, -117, \\
& -145, -119, -46, 46, 122, 153, \\
& 125, 48, -48, -128, -159, -130, \\
& -50, 51, 133, 166, 136, 52, \\
& -53, -139, -173, -141, -54, 55, \\
& 144, 180, 146, 56, -57, -149, \\
& -186, -151, -58, 59, 155, 192, \\
& 156, 60, -61, -159, -198, -161, \\
& -62, 62, 164, 204, 166, 64, \\
& -64, -168, -209, -170, -65, 66, \\
& 173, 214, 174, 67, -67, -177, \\
& -219, -178, -68, 69, 180, 224, \\
& 182, 70, -70, -184, -228, -185, \\
& -71, 71, 187, 231, 188, 72, \\
& -72, -189, -235, -190, -73, 73, \\
& 192, 238, 193, 74, -74, -194, \\
& -240, -195, -74, 75, 196, 242, \\
& 196, 75, -75, -197, -244, -197, \\
& -75, 76, 198, 245, 198, 76, \\
& -76, -199, -246, -199, -76, 76, \\
& 199, 246, 199, 76, -76, -199, \\
& -246, -199, -76, 76, 198, 245, \\
& 198, 76, -75, -197, -244, -197, \\
& -75, 75, 196, 242, 196, 75, \\
& -74, -195, -240, -194, -74, 74, \\
& 193, 238, 192, 73, -73, -190, \\
& -235, -189, -72, 72, 188, 231, \\
& 187, 71, -71, -185, -228, -184, \\
& -70, 70, 182, 224, 180, 69, \\
& -68, -178, -219, -177, -67, 67, \\
& 174, 214, 173, 66, -65, -170, \\
& -209, -168, -64, 64, 166, 204, \\
& 164, 62, -62, -161, -198, -159, \\
& -61, 60, 156, 192, 155, 59, \\
& -58, -151, -186, -149, -57, 56,
\end{aligned}
$$

$$
\begin{array}{rrrrrr}
146, & 180, & 144, & 55, & -54, & -141, \\
-173, & -139, & -53, & 52, & 136, & 166, \\
133, & 51, & -50, & -130, & -159, & -128, \\
-48, & 48, & 125, & 153, & 122, & 46, \\
-46, & -119, & -145, & -117, & -44, & 44, \\
113, & 138, & 111, & 42, & -41, & -107, \\
-131, & -105, & -40, & 39, & 102, & 124, \\
99, & 38, & -37, & -96, & -117, & -94, \\
-35, & 35, & 90, & 110, & 88, & 33, \\
-33, & -85, & -103, & -82, & -31, & 31, \\
79, & 96, & 77, & 29, & -29, & -74, \\
-90, & -72, & -27\} & & &
\end{array}
$$

# Anhang C: CD-Inhalte

Auf der beiliegenden CD befinden sich folgende Inhalte:

- PDF-Version der Arbeit
- Quellcode der Firmware inklusive aller Bibliotheken und KEIL $\mu$Vision-Projektdateien
- Quellcode der Dekstopanwendung inklusive Projektdateien für Visual Studio 2015
- Schaltpläne der erstellten Hardware für CadSoft EAGLE bzw. Autodesk EAGLE
- Handbücher der verwendeten Hardware

# Literaturverzeichnis

[1] J. BLAUERT, *Vorlesungsscript Akustik 2*. https://upload.wikimedia.org/wikipedia/commons/7/78/Akustik_Mithoerschwelle2.JPG, 2004. Ruhr-Universität Bochum.

[2] BUNDESANSTALT FÜR DEN DIGITALFUNK DER BEHÖRDEN UND ORGANISATIONEN MIT SICHERHEITSAUFGABEN, *Das Projekt - Meilensteine in der Einführung des Digitalfunk BOS*. http://www.bdbos.bund.de/DE/Digitalfunk_BOS/Meilensteine/projekt_meilensteine_node.html, 2013.

[3] CRYPTOOL-PROJEKT. www.cryptool.org, 2008.

[4] DEUTSCHE TELEKOM AG, *Technische Beschreibung der analogen Wählanschlüsse am Netz der Deutschen Telekom, 1 TR 110 - 1, Telefonanschlüsse ohne Durchwahl*. https://www.telekom.de/hilfe/downloads/1tr110-1_ausgabe_11-2015_v12.pdf.

[5] ELEKTRONIKLADEN, *Benutzerhandbuch - chip1768 v1.21*.

[6] ELEKTRONIKLADEN, *Benutzerhandbuch - testbed for mbed / mbed entwicklungssystem*.

[7] GEORG AND OTFRIED, *Telekommunikationstechnik: Handbuch für Lehre und Praxis*, Springer, 2000.

[8] G. HABERMANN, *Zur spektrometrischen Analyse des bewegten Klanges*.

[9] INFORMATION SCIENCES INSTITUTE, UNIVERSITY OF SOUTHERN CALIFORNIA, *RFC: 793 - TRANSMISSION CONTROL PROTOCOL, DARPA INTERNET PROGRAM, PROTOCOL SPECIFICATION*.

[10] N. JAYANT, B. MCDERMOTT, S. CHRISTENSEN, AND A. QUINN, *A comparison of four methods for analog speech privacy*, IEEE TRANSACTIONS ON COMMUNICATIONS, (1981).

[11] J. KAISER AND R. SCHAFER, *On the use of the io -sinh window for spectrum analysis*, IEEE TRANSACTIONS ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING, VOL. ASSP-2, NO. 1, FEBRUARY 1980, (1980).

[12] H. KOHAD, P. V. INGLE, AND DR.M.A.GAIKWAD, *An Overview of Speech Encryption Techniques*. http://www.ijerd.com/paper/vol3-issue4/E03042932.

pdf, 2012.

[13] E. KOHLER, *Buffered I/O and Synchronization I: Sequential consistency*. http://www.read.cs.ucla.edu/111/notes/lec7, 2011. University of California, Computer Science.

[14] R. LERCH, *Elektrische Messtechnik*, Springer, 2005.

[15] M. MEYER, *Kommunikationstechnik*, Vieweg, 1999.

[16] MICROCHIP TECHNOLOGY INCORPORATED, *MCP3001 - 2.7V 10-Bit A/D Converter with SPI™ Serial Interface*, 12 2006.

[17] NXP B.V., *UM10360 - LPC17xx User manual (Rev. 2)*.

[18] PH.D. S.C. B.E. KAK , *Overview of analogue signal encryption*, IEE PROCEEDINGS, Vol. 130, Pi. F, No. 5, (1983).

[19] M. REUTER, *Telekommunikation*, Deckers Verlag, 1990.

[20] R. P. SALLEN AND E. L. KEY, *A practical method of designing rc active filters*, IRE Transactions on Circuit Theory ( Volume: 2, Issue: 1, March 1955 ), (1955).

[21] W. SARAGA, *SENSITIVITY OF 2nd-ORDER SALLEN-KEY-TYPE ACTIVE RC FILTERS*.

[22] N. S. SHUSHANK DOGRA, *Comparison of different techniques to design of filter*, International Journal of Computer Applications, Volume 97– No.1, July 2014, (2014).

[23] SILICON LABS, *AN118 - IMPROVING ADC RESOLUTION BY OVERSAMPLING AND AVERAGING*. http://www.silabs.com/Support%20Documents/TechnicalDocs/an118.pdf.

[24] P. TEEHAN, M. GREENSTREET, AND G. LEMIEUX, *A Survey and Taxonomy of GALS Design Styles*. http://www.ece.ubc.ca/~lemieux/publications/teehan-ieeedtcomputer2007.pdf. IEEE Design and Test of Computers.

[25] TEXAS INSTRUMENTS, *Op amps for everyone*, 2008. Active Filter Design.

[26] T. WÜBBE, *Telekommunikationstechnik*, Vogel Fachbuch, 2010.

[27] M. WERNER, *Nachrichtentechnik*, Springer, 2009.

[28] ZOLLNER AND ZWICKER, *Elektroakustik*, Springer, 1998.

# Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 27. März 2017